LEARN TEACH

Designing a Web Crawler

Let's design a Web Crawler that will systematically browse and download the World Wide Web. Web crawlers are also known as web spiders, robots, worms, walkers, and bots.

Difficulty Level: Hard

1. What is a Web Crawler?

A web crawler is a software program which browses the World Wide Web in a methodical and automated manner. It collects documents by recursively fetching links from a set of starting pages. Many sites, particularly search engines, use web crawling as a means of providing up-to-date data. Search engines download all the pages to create an index on them to perform faster searches.

Some other uses of web crawler are:

LEARN TEACH

- To monitor sites to see when their structure or contents change.
- To maintain mirror sites for popular Web sites.
- To search for copyright infringements.
- To build a special-purpose index, e.g., one that has some understanding of the content stored in multimedia files on the Web.

2. Requirements and Goals of the System

Let's assume we need to crawl all the web.

Scalability: Our service needs to be scalable such that it can crawl the entire Web, and can be used to fetch hundreds of millions of Web documents.

Extensibility: Our service should be designed in a modular way, with the expectation that new functionality will be added to it. There could be newer document types that needs to be downloaded and processed in the future.

3. Some Design Considerations

Crawling the web is a complex task, and there are many issues we need to think through before coming up with a workable design. Here are some of the considerations:

LEARN TEACH

Is it a crawler for HTML pages only? Or should we fetch and store other types of media, such as sound files, images, videos, etc.? This is important because the answer can change the design. If we are writing a general-purpose crawler to download different media types, we might want to break down the parsing module into different sets of modules: one for HTML, another for images, another for videos, where each module extracts what is considered interesting for that media type.

Let's assume for now that our crawler is going to deal with HTML only, but it should be extensible and make it easy to add support for new media types.

What protocols are we looking at? HTTP? What about FTP links? What different protocols should our crawler handle? For the sake of the exercise, we will assume HTTP. Again, it shouldn't be hard to extend the design to use FTP and other protocols later.

What is the expected number of pages we will crawl? How big will the URL database become? Assuming we need to crawl one billion websites. Since a website can contain many, many URLs, let's assume an upper bound of 15 billion different web pages that will be reached by our crawler.

What is 'RobotsExclusion' and how should we deal with it? Courtesous Web crawlers implement the Robots Exclusion Protocol, which means that there is a file on each Web site, called robots.txt, that dictates how crawlers can access the site. For example, a site may disallow all crawlers from a certain directory, or may disallow a specific crawler from accessing any of its URLs. Our crawler will need to parse such files and understand the various directives they may contain before downloading any URL from a website.

LEARN TEACH

4. Capacity Estimation and Constraints

If we want to crawl 15 billion pages within four weeks, how many pages do we need to fetch per second?

$$15B / (4 \text{ weeks} * 7 \text{ days} * 86400 \text{ sec}) \approx 6200 \text{ pages/sec}$$

What about storage? Page sizes vary a lot, but as mentioned above since we will be dealing with HTML text only, let's assume an average page size be 100KB. With each page if we are storing 500 bytes of metadata, total storage we would need:

$$15B * 100KB + 500 \approx 1.5 \text{ petabytes}$$

Assuming a 70% capacity model (we don't want to go above 70% of the total capacity of our storage system), total storage we will need:

$$1.5 \text{ petabytes} / 0.7 \approx 2.14 \text{ petabytes}$$

5. High Level design

LEARN TEACH

Following steps.

1. Pick a URL from the unvisited URL list.
2. Determine the IP Address of its host-name.
3. Establishing a connection to the host to download the corresponding document.
4. Parse the document contents to look for new URLs.
5. Add the new URLs to the list of unvisited URLs.
6. Process the downloaded document, e.g., store it or index its contents, etc.
7. Go back to step 1

How to crawl?

Breadth-first or depth-first? Breadth-first search (BFS) is usually used. However, Depth First Search (DFS) is also utilized in some situations, such as if your crawler has already established a connection with the website, it might just DFS all the URLs within this website to save some handshaking overhead.

Path-ascending crawling: Path-ascending crawling can help discover a lot of isolated resources or resources for which no inbound link would have been found in regular crawling of a particular Web site. In this scheme, a crawler would ascend to every path in each URL that it intends to crawl. For example, when given a seed URL of http://foo.com/a/b/page.html, it will

LEARN TEACH

Difficulties in implementing efficient web crawler

There are two important characteristics of the Web that makes Web crawling a very difficult task:

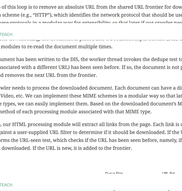
1. Large volume of Web pages: A large volume of web page implies that web crawler can only download a fraction of the web pages at any time and hence it is critical that web crawler should be intelligent enough to prioritize download.

2. Rate of change on web pages. Another problem with today's dynamic world is that web pages on the internet change very frequently, as a result, by the time the crawler is downloading the last page from a site, the page may change, or a new page has been added to the site.

A bare minimum crawler needs at least these components:

1. URL Frontier: To store the list of URLs to download and also prioritize which URLs should be crawled first.
2. HTTP Fetcher: To retrieve a web page from the server.
3. Extractor: To extract links from HTML documents.
4. Duplicate Eliminator: To make sure same content is not extracted twice unintentionally.
5. Datastore: To store retrieve pages and URL and other metadata.

LEARN TEACH



6. Detailed Component Design

Let's assume our crawler is running on one server, and all the crawling is done by multiple working threads, where each working thread performs all the steps needed to download and process a document in a loop.

The first step of this loop is to remove an absolute URL from the shared URL frontier for downloading. An absolute URL begins with a scheme (e.g., "HTTP"), which identifies the network protocol that should be used to download it. We can implement these protocols in a modular way for extensibility, so that later if our crawler needs to support more protocols, it can easily add more modules. Based on the URL's scheme, the working thread finds the protocol module to download the document. For instance, these modules could also have mechanisms to impose restrictions on what URLs they can download. For example, we can restrict our crawler from downloading any URLs from certain IP addresses or domains, or we can restrict our crawler from downloading any URL that has a "gov" or "mil" extension. These restrictions can be made by either parsing and processing robots.txt files or by maintaining a blacklist of IP addresses and URL domains. We can also use these modules to impose some restrictions on our crawler to avoid any unintentional or malicious attacks on other sites. For instance, our crawler can limit the maximum number of HTTP requests to any single server, or it may be required to wait a certain amount of time between two HTTP requests to a single server. We can easily implement these extensibility options in our URL extraction module, so that if later we need to impose more restrictions, we can easily add them to this module.

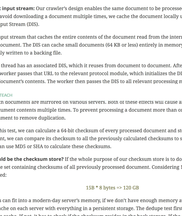
LEARN TEACH

Once the document has been written to the DIS, the worker thread invokes the dedupe test to determine whether this document (associated with a different URL) has been seen before. If so, the document is not processed any further, and the worker thread removes the next URL from the frontier.

Next, our crawler needs to process the downloaded document. Each document can have a different MIME type like HTML, page, Image, Video, etc. We can implement these MIME schemes in a modular way so that later if our crawler needs to support more types, we can easily implement them. Based on the downloaded document's MIME type, the worker invokes the process method of each processing module associated with that MIME type.

Furthermore, our HTML processing module will extract all links from the page. Each link is converted into an absolute URL and tested against a user-supplied URL filter to determine if it should be downloaded. If the URL passes the filter, the worker performs the URL-seen test, which checks if the URL has been seen before, namely, if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier.

LEARN TEACH



LEARN TEACH

4. URL Dedupe Test: Text URL extracted in our web scraper may contain many URLs that point to the same content, and so the URL frontier may contain many URLs that point to the same web page. We need to prevent our crawler from adding such URLs to the URL frontier, otherwise our crawler would end up in a tight loop downloading the same URL over and over again. We can accomplish this by testing each URL we extract against a list of URLs we have seen before. If the URL is not in the list, we add it to the list and also add it to the URL frontier; otherwise, we do not add it to the URL frontier.

Because billions of URLs may have been downloaded over the years, looking up each URL in the list to see if it has been seen before can be time-consuming and slow. Also, keeping such a huge list in memory may be prohibitively expensive. There are also other scalability issues that we will discuss later.

LEARN TEACH

5. URL Distribution: The URL frontier is the data structure that contains all the URLs that are to be downloaded. We can crawl by performing a breadth-first traversal of the Web, starting from the pages in the seed set. Such traversals are easily implemented by using a FIFO queue.

Since we'll be having a huge list of URLs to crawl, we can distribute our URL frontier into multiple servers. Let's assume on each server we have multiple worker threads performing the crawling tasks. Let's also assume that our hash function maps each URL to a server which will be responsible for crawling it.

Following politeness requirements must be kept in mind while designing a distributed URL frontier:

1. Our crawler should not overload a server by downloading a lot of pages from it.
2. We should not have multiple machines connecting a web server.

To implement this politeness constraint, our crawler can have a collection of distinct FIFO sub-queues on each server. 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. Our hash function can map each hostname to a thread number. Together, these two points imply that at most one worker thread will download documents from a given Web server and also by using FIFO queue it'll not overload a Web server.

LEARN TEACH

6. Fault Tolerance: We should be able to handle machine failures and keep our crawl going. Crawling a huge number of URLs will span many days. It is quite possible that a worker thread may be slow or stuck, or a machine may crash, or may be down for some time. We should be able to restore the crawl state and data structures of our crawler from the last checkpoint. We can also restart the failed worker threads on other servers.

7. Timeout: Some URLs may take a long time to respond, or may never respond to our HTTP requests. To ensure that we don't wait for ever, we can specify a timeout before which we will not wait for a response from a web server. Many URLs will be slow, and we can't wait for all of them before we start downloading other URLs.

LEARN TEACH

8. Spam Protection: Our crawler should be able to identify and avoid spam sites. Spam can be in many forms, and we can start with a few simple rules to identify such websites.

9. Extensibility: Our crawler should be designed in a modular way, with the expectation that new functionality will be added to it. There could be newer document types that needs to be downloaded and processed in the future.

LEARN TEACH

10. Canonicalization: A canonical form is a standard way of denoting a URL. There are many variations of a URL that can point to the same content. For instance, the URL www.google.com may be the same as www.google.com/index.htm. Our URL processing module should do some simple URL canonicalization before inserting a URL into the URL frontier.

11. Dedupe Test: While extracting links, any Web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a URL dedupe test must be performed on each extracted link before adding it to the URL frontier.

To perform the URL dedupe test, we can store all the URLs seen by our crawler in canonical form in a database. To save space, we do not store the textual representation of each URL in the URL set, but rather a fixed-sized checksum.

To reduce the number of operations on the database store, we can keep an in-memory cache of popular URLs on each host shared by all threads. The reason to have this cache is that links to some URLs are quite common, so caching the popular ones in memory will lead to a high in-memory hit rate.

How much storage we would need for URL's store? If the whole purpose of our checksum is to do URL dedupe, then we

LEARN TEACH

$$15B * 2 \text{ bytes} => 30 \text{ GB}$$

Can we use bloom filters for deduping? Bloom filters are a probabilistic data structure for set membership testing that may yield false positives. A large bit vector represents the set. An element is added to the set by computing 'n' hash functions of the element and setting the corresponding bits. An element is deemed to be in the set if the bits at all 'n' of the element's hash locations are set. Hence, a document may incorrectly be deemed to be in the set, but false negatives are not possible.

The disadvantage to using a bloom filter for the URL seen test is that each false positive will cause the URL not to be added to the frontier, and therefore the document will never be downloaded. The chance of a false positive can be reduced by making the bit vector larger.

8. Checkpointing: A crawl of the entire Web takes weeks to complete. To guard against failures, our crawler can write regular snapshots of its state to disk. An interrupted or aborted crawl can easily be restarted from the latest checkpoint.

7. Fault tolerance

We should use consistent hashing for distribution among crawling servers. Extended hashing will not only help in replacing

LEARN TEACH

dead hosts or hosts that are not responding, but it will also help in distributing load among the crawlers. Consistent hashing will also make it much easier to add new machines to the crawl cluster when we need to speed up our crawl. Since we are using consistent hashing, if one of the servers goes down, we can replace it. Meanwhile, extended hashing should shift the load to other servers.

8. Data Partitioning

Our crawler will be dealing with three kinds of data: 1) URLs to visit 2) URLs checksums for dedupe 3) Document checksums for dedupe.

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 extended hashing, we can assume that URLs will be redistributed from overloaded hosts.

Each host will perform checkpointing periodically and dump a snapshot of all the data it is holding into a remote server. This will ensure that if a server dies down, another server can replace it by taking its data from the last snapshot.

9. Crawler Traps

There are many crawler traps, spam sites, and cloaked content. A crawler trap is a URL or set of URLs that cause a crawler to crawl indefinitely. Some crawler traps are unintentional. For example, a symbolic link within a file system can create a

LEARN TEACH

trap, but most others are malicious, and intended to slow down a crawler. For example, a CGI script can keep generating a page with a new URL and infinite amount of data, or a page may contain a huge amount of meaningless data. Many spam sites are designed to confuse and trick a crawler by spamming keywords or irrelevant content. Such sites may also contain many URLs that point to other spam sites.

Some of these crawler traps can be avoided if we can maintain a list of URL patterns that we should not follow. We can also manually identify such sites and put their URL patterns in our URL filter to avoid them in future.

LEARN TEACH

10. Prioritizing Pages

Our crawler should be able to prioritize which pages to download first. We can prioritize based on many factors such as popularity, PageRank, URL length, URL prefix, etc. For example, we may decide to give low priority to URLs with very long hostnames or URLs with many tokens. We can also give low priority to URLs that are very deep into a website's hierarchy, as such URLs may be less important than URLs that are closer to the root directory of a website.

LEARN TEACH

11. Prioritizing Crawling

Let's assume we have billions of pages in our document store, and we've crawled a billion URLs based on the URLs we have collected from the pages we crawled so far. We can use the PageRank of these pages to decide which pages should be crawled next; we can prioritize URLs that point to more popular sites. We can also use our crawl history to estimate how often a page changes on the Web, and we can use this information to prioritize URLs that are more stale.

LEARN TEACH

12. Conclusion

Designing a web crawler is a very complex task. There are many issues we haven't discussed here, such as URL caching, SEO ranking, Web traffic, and many Web protocols. We also haven't discussed the ranking of the crawled documents, or the crawl strategy to "crawl the most important pages first", or how to update documents that have been crawled earlier, etc.

LEARN TEACH

13. Further Reading

Mercator: A scalable, extensible Web crawler

Web Crawling

Web Crawler Beyond Frontiers

Efficient URL Caching for World Wide Web Crawling

Crawling the Web

Web Crawler Architecture

Mark as completed

← Previous
Designing Twitter Search

Next
Designing Facebook's Newsfeed

Send feedback or ask a question · 18 recommendations