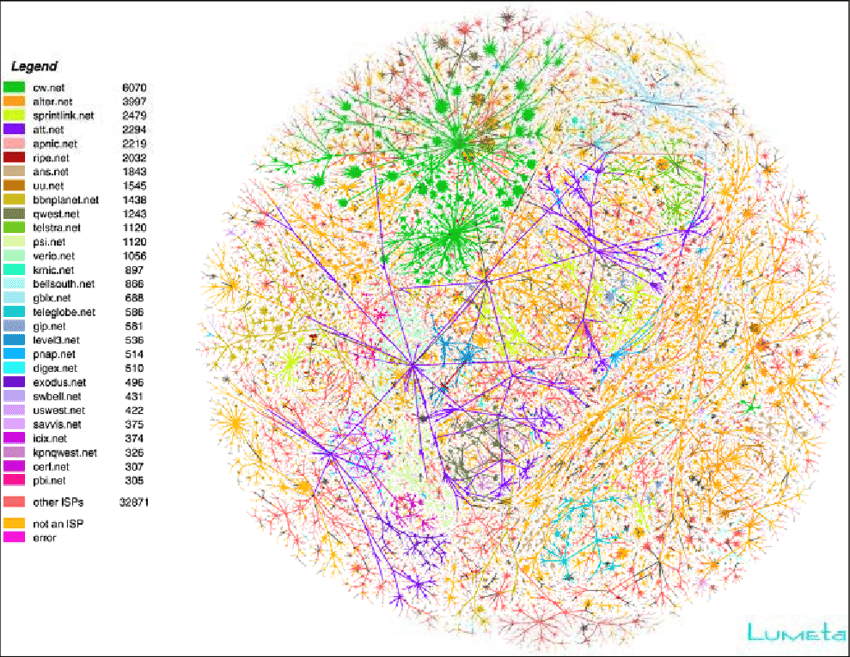
**Web Crawler | System Design Interview Question**

**Use case**

The internet is a HUUUGE graph with web pages being its nodes and hyperlinks being its edges. A web crawler(spider) is a bot that traverses the internet graph.

This is a graph of the internet created 20 years ago:



There are many purposes of a web crawler. For example, search engines like Google use web crawlers to create web indices. The web index created by the web crawler is typically an inverted index, i.e. a mapping from content to document id. Imagine we have three documents:

* [website1.com/a:](http://website1.com/a:) "system design interview"
* [website2.com/b:](http://website2.com/b:) "algorithm interview"
* [website3.com/c:](http://website3.com/c:) "tech interview system"

The inverted index would look like this

1{

2 "system": ["website1.com/a", "website3.com/c"],

3 "design": ["website1.com/a"],

4 "interview": ["website1.com/a", "website2.com/b", "website3.com/c"],

5 "algorithm": ["website2.com/a"],

6 "tech": ["website3.com/a"],

7}

It should be obvious an inverted index provides quick access to document ids that contain a term. Now if a user searches for "system interview", we can find the intersection of the "system" set and "interview" set.

The web crawler for search engines could be much more complex than our example. If you are interested, you can read [Google's patent](https://patents.google.com/patent/US20110307467A1/en) on web crawler architecture.

For simplicity, we will develop a web crawler for mirroring website by download all html pages of the given website, with [real demo code](https://github.com/realAlgoMonster/system-design/tree/main/web-crawler).

**Back of the envelop calculations**

Let's do some estimation and calculation. Suppose we want to fetch 1 billion HTML document per year on average, then we have 1000/365≈3 million new files per day, and 3000000/86400≈35 requests per second.

According to the [HTTP Archive](https://httparchive.org/reports/page-weight#bytesHtml), the average size of HTML documents is 30KB, then we can store about 35 thousand links in 1GB storage. Hence we need about 30TB storage per year.

**Service**

At a high level, a web crawler has to

1. Download a page
2. Create and update the index (if the goal is to index and rank pages) or write the page to filesystem (if the goal is to mirror a site)
3. Extract links and check duplicates
4. Visit pages linked from the current page

**DFS vs BFS**

Since the problem we are trying to solve is essentially graph traversal, it’s natural to ask whether to use [Depth-first search](https://www.notion.so/problems/dfs_intro) or [Breadth-first search](https://www.notion.so/problems/bfs_intro). If the internet were static and we have no time requirement, both algorithm have the same complexity. In reality, if the resource is limited and we are trying to crawl as many websites as possible, then we would crawl only the most important pages. In most case, this is the home page of each website. Under these constraints, BFS would be superior to DFS.

Does that mean DFS is not used? Not really. To crawl a website, the crawler has to establish a TCP connection which requires expensive setups such as the three handshakes. It’s a bit of a waste to only visit the home page in one connection. In this case, it’s more efficient to visit all the pages of a website in one TCP connection, which is essentially DFS.

In reality, all websites are not created equal, the order of crawling is managed by a Scheduler. A scheduler store the URLs to be crawled in a Priority Queue. Overall, a web crawling process is more of BFS than DFS.

Now that we have clarify the problem and explored the potential solutions, we can propose a high-level design:

Let's look at each components:

**Downloader**

Downloader is the worker(s) that take a url and downloads the HTML file and saves it into storage and also sends the HTML body to Link Extractor to extract the links in the HTML. It pops tasks (which contains the url to be fetched) from the task queue.

**Task Queue**

Task Queue is a message queue of urls to crawl. Our workers (Downloader) fetch the message from the task queue.

**Link Extractor**

Link Extractor extracts links ([anchor tags](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a)) and pushes them into the Task Queue if the url hasn't been crawled yet.

**Finished URLs**

This is the algorithm equivalent of a hashset that stores all the URLs that have been crawled.

**Storage**

From our calculations, we need about 30TB storage per year, so the local file system is probably not enough. For such large projects, we can use distributed file storage like HDFS, or tools like [Rclone](https://rclone.org/) to mount cloud storage like AWS S3 onto the file system and have the downloader write directly into them.

**Sample Implementation**

"Talk is cheap. Show me the code. - Linus Torvalds"

To help you understand the idea better, we show a sample implementation of mirroring a website. We will download the entire website of <http://quotes.toscrape.com/> and save it in our local file system.

It's very common to use Python for web crawlers, and [Scrapy](https://scrapy.org/) is a very popular web crawlering framework with high quality [documentation](https://docs.scrapy.org/) and [tutorial](https://docs.scrapy.org/en/latest/intro/tutorial.html). Scrapy is powerful and extensible, and it can handle robots.txt for you.

For the queue design, we can use a in-memory data structure since the items in the queue are just transient work items and do not need to be persisted. We will use Redis's list data type. You can read more about using Redis as a queue in the official [Redis documentation](https://redis.com/ebook/part-2-core-concepts/chapter-6-application-components-in-redis/6-4-task-queues/6-4-1-first-in-first-out-queues/). Alternatively, you can use [Kafka](https://kafka.apache.org/) or [RabbitMQ](https://www.rabbitmq.com/). However, we don't need data persistence and our model is very simple, so Redis is our choice.

Redis also doubles as a distributed hashset to store finished links here.

Here is the code of our spider.

1# spider.py

2import scrapy

3from urllib.parse import urlparse

4from scrapy.linkextractors import LinkExtractor

5import redis

6import os

7import time

8from pathlib import Path

9

10class QuotesSpider(scrapy.Spider):

11 name = "quotes"

12 le = LinkExtractor()

13 r = redis.Redis(host='localhost', port=6379, db=0)

14 path = 'data'

15 hostname = ''

16

17 def start\_requests(self):

18 # init task queue with a URL

19 self.r.rpush('task\_queue', 'http://quotes.toscrape.com/page/1/')

20

21 while True:

22 item = self.r.lpop('task\_queue')

23 if item is None:

24 time.sleep(1)

25 item = self.r.lpop('task\_queue')

26 if item is None:

27 break

28 url = item.decode("utf-8")

29 if self.hostname == '':

30 self.hostname = urlparse(url).hostname

31 yield scrapy.Request(url=url, callback=self.parse)

32

33 def parse(self, response):

34 filepath = urlparse(response.url).path

35 if filepath[-1] == '/':

36 filepath += 'index.html'

37 path = self.path + filepath

38 Path(os.path.dirname(path)).mkdir(parents=True, exist\_ok=True)

39 with open(path, 'wb') as f:

40 f.write(response.body)

41

42 self.r.sadd('task\_finished', response.url)

43

44 for link in self.le.extract\_links(response):

45 url = link.url

46 if urlparse(url).hostname != self.hostname:

47 break

48 if self.r.sismember('task\_finished', url):

49 break

50 self.r.rpush('task\_queue', url)

The web crawler continuously pop urls from the front of the task queue, which is a redis list. After write the HTML document to the file system, the crawler adds the url to the set of finished urls, extract the urls in the current HTML document and check if a url is already finished. If not, then the crawler pushes the url to the end of the task queue.

You can clone the code on our [GitHub](https://github.com/realAlgoMonster/system-design/tree/main/web-crawler).

To run these code on your machine, you need to have python3 and pip installed. For Redis, you can [install](https://redis.io/topics/quickstart) on your machine, or use docker.

1# install python dependency

2pip install scrapy redis

3

4# run redis from docker

5docker run --name web-crawler-redis -p 6379:6379 -d redis

6

7# start crawling

8scrapy runspider spider.py

This spider can crawl [quotes.toscrape.com](http://quotes.toscrape.com) in one second.

12021-10-09 16:29:24 [scrapy.core.engine] INFO: Closing spider (finished)

22021-10-09 16:29:24 [scrapy.statscollectors] INFO: Dumping Scrapy stats:

3{'downloader/request\_bytes': 18910,

4'downloader/request\_count': 56,

5'downloader/request\_method\_count/GET': 56,

6'downloader/response\_bytes': 85706,

7'downloader/response\_count': 56,

8'downloader/response\_status\_count/200': 48,

9'downloader/response\_status\_count/308': 8,

10'dupefilter/filtered': 4,

11'elapsed\_time\_seconds': 3.39268,

12'finish\_reason': 'finished',

13'finish\_time': datetime.datetime(2021, 10, 9, 8, 29, 24, 940764),

14'httpcompression/response\_bytes': 305900,

15'httpcompression/response\_count': 48,

16'log\_count/DEBUG': 57,

17'log\_count/INFO': 10,

18'memusage/max': 55025664,

19'memusage/startup': 55025664,

20'response\_received\_count': 48,

21'scheduler/dequeued': 56,

22'scheduler/dequeued/memory': 56,

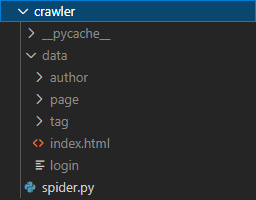
23'scheduler/enqueued': 56,

24'scheduler/enqueued/memory': 56,

25'start\_time': datetime.datetime(2021, 10, 9, 8, 29, 21, 548084)}

262021-10-09 16:29:24 [scrapy.core.engine] INFO: Spider closed (finished)

And the entire website has been crawled into data folder.



[Go Premium](https://algo.monster/subscribe)