## Step 2 - Propose high-level design and get buy-in

In this section, we discuss the API endpoints, URL redirecting, and URL shortening flows.

## API Endpoints

API endpoints facilitate the communication between clients and servers. We will design the APIs REST-style. If you are unfamiliar with restful API, you can consult external materials, such as the one in the reference material [1]. A URL shortener primary needs two API endpoints.

1.URL shortening. To create a new short URL, a client sends a POST request, which contains one parameter: the original long URL. The API looks like this:

*POST api/v1/data/shorten*

- request parameter: {longUrl: longURLString}
- return shortURL

2.URL redirecting. To redirect a short URL to the corresponding long URL, a client sends a GET request. The API looks like this:

*GET api/v1/shortUrl*

- Return longURL for HTTP redirection

## URL redirecting

Figure 8-1 shows what happens when you enter a tinyurl onto the browser. Once the server receives a tinyurl request, it changes the short URL to the long URL with 301 redirect.


Figure 8-1

The detailed communication between clients and servers is shown in Figure 8-2.

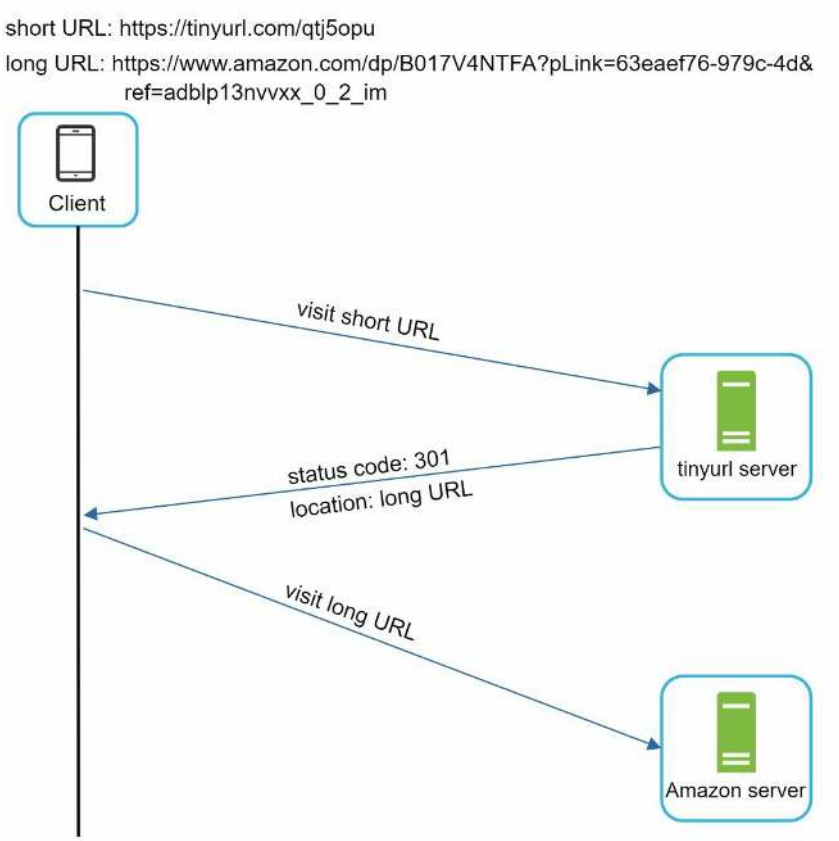


Figure 8-2

One thing worth discussing here is 301 redirect vs 302 redirect.

**301 redirect**. A 301 redirect shows that the requested URL is "permanently" moved to the long URL. Since it is permanently redirected, the browser caches the response, and subsequent requests for the same URL will not be sent to the URL shortening service. Instead, requests are redirected to the long URL server directly.

**302 redirect**. A 302 redirect means that the URL is "temporarily" moved to the long URL, meaning that subsequent requests for the same URL will be sent to the URL shortening service first. Then, they are redirected to the long URL server.

Each redirection method has its pros and cons. If the priority is to reduce the server load, using 301 redirect makes sense as only the first request of the same URL is sent to URL shortening servers. However, if analytics is important, 302 redirect is a better choice as it can track click rate and source of the click more easily.

The most intuitive way to implement URL redirecting is to use hash tables. Assuming the hash table stores <*shortURL, longURL*> pairs, URL redirecting can be implemented by the following:

- Get longURL: longURL = hashTable.get(shortURL)
- Once you get the longURL, perform the URL redirect.

## URL shortening

Let us assume the short URL looks like this: www.tinyurl.com/**{hashValue}**. To support the URL shortening use case, we must find a hash function *fx* that maps a long URL to the *hashValue*, as shown in Figure 8-3.

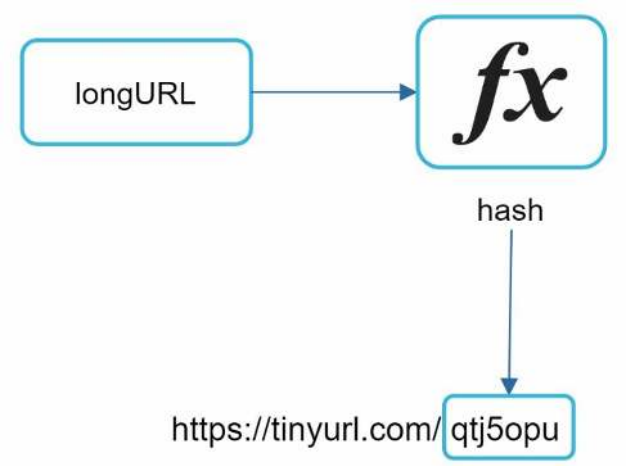


Figure 8-3

The hash function must satisfy the following requirements:

- Each *longURL* must be hashed to one *hashValue*.
- Each *hashValue* can be mapped back to the *longURL*.

Detailed design for the hash function is discussed in deep dive.

## Step 3 - Design deep dive

Up until now, we have discussed the high-level design of URL shortening and URL redirecting. In this section, we dive deep into the following: data model, hash function, URL shortening and URL redirecting.

## Data model

In the high-level design, everything is stored in a hash table. This is a good starting point; however, this approach is not feasible for real-world systems as memory resources are limited and expensive. A better option is to store <*shortURL, longURL*> mapping in a relational database. Figure 8-4 shows a simple database table design. The simplified version of the table contains 3 columns: *id, shortURL, longURL*.

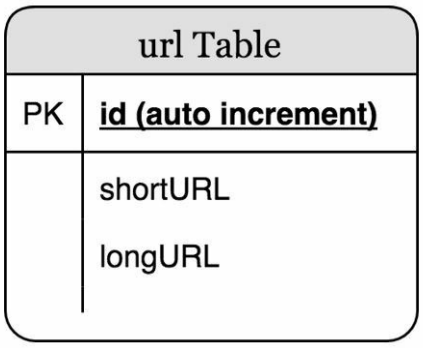


Figure 8-4

## Hash function

Hash function is used to hash a long URL to a short URL, also known as *hashValue*.

### Hash value length

The *hashValue* consists of characters from [0-9, a-z, A-Z], containing 10 + 26 + 26 = 62 possible characters. To figure out the length of *hashValue*, find the smallest *n* such that *62^n ≥ 365 billion*. The system must support up to 365 billion URLs based on the back of the envelope estimation. Table 8-1 shows the length of *hashValue* and the corresponding maximal number of URLs it can support.

| N | Maximal number of URLs |
|---|---|
| 1 | 62^1 = 62 |
| 2 | 62^2 = 3,844 |
| 3 | 62^3 = 238,328 |
| 4 | 62^ 4 = 14,776,336 |
| 5 | 62^5 = 916,132,832 |
| 6 | 62^6 = 56,800,235,584 |
| 7 | 62^7 = 3,521,614,606,208 = ~3.5 trillion |

Table 8-1

When *n = 7, 62 ^ n = ~3.5 trillion*, 3.5 trillion is more than enough to hold 365 billion URLs, so the length of *hashValue* is 7.

We will explore two types of hash functions for a URL shortener. The first one is “hash + collision resolution”, and the second one is “base 62 conversion.” Let us look at them one by one.

**Hash + collision resolution**

To shorten a long URL, we should implement a hash function that hashes a long URL to a 7-character string. A straightforward solution is to use well-known hash functions like CRC32, MD5, or SHA-1. The following table compares the hash results after applying different hash functions on this URL: https://en.wikipedia.org/wiki/Systems_design.

| **Hash function** | **Hash value (Hexadecimal)** |
|---|---|
| CRC32 | 5cb54054 |
| MD5 | 5a62509a84df9ee03fe1230b9df8b84e |
| SHA-1 | 0eeae7916c06853901d9ccbefbfcaf4de57ed85b |

Table 8-2

As shown in Table 8-2, even the shortest hash value (from CRC32) is too long (more than 7 characters). How can we make it shorter?

The first approach is to collect the first 7 characters of a hash value; however, this method can lead to hash collisions. To resolve hash collisions, we can recursively append a new predefined string until no more collision is discovered. This process is explained in Figure 8-

5.

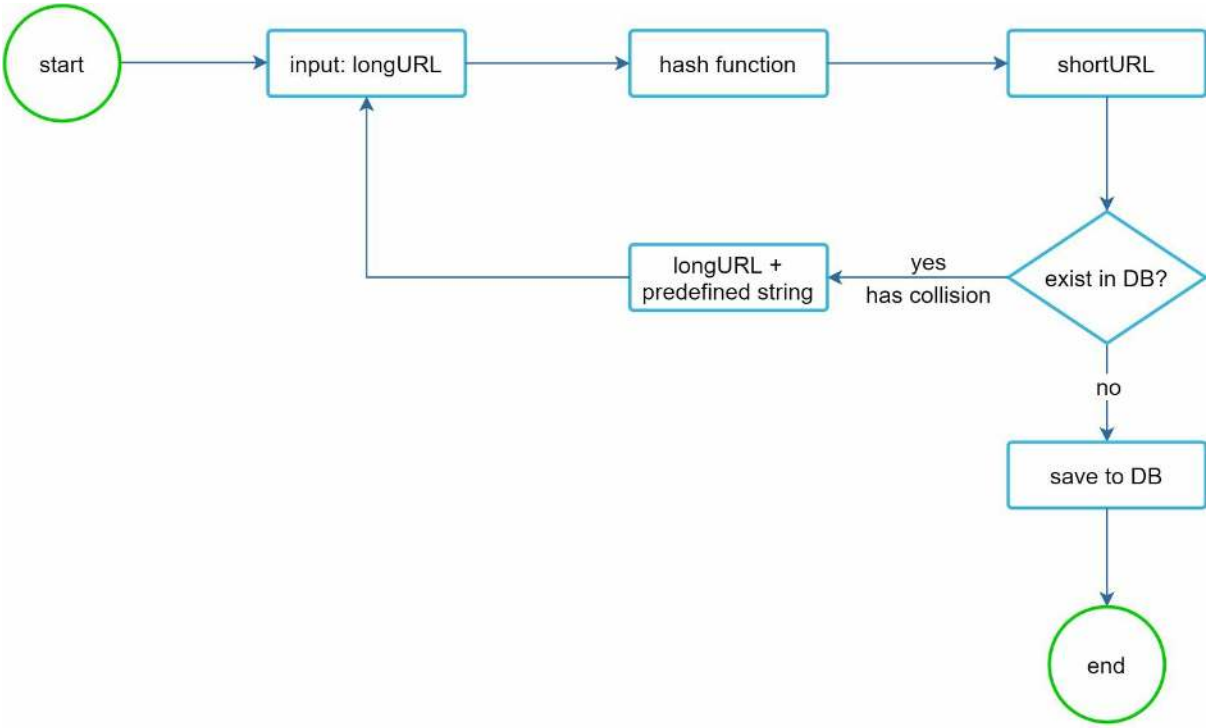


Figure 8-5

This method can eliminate collision; however, it is expensive to query the database to check if a shortURL exists for every request. A technique called bloom filters [2] can improve performance. A bloom filter is a space-efficient probabilistic technique to test if an element is a member of a set. Refer to the reference material [2] for more details.

**Base 62 conversion**

Base conversion is another approach commonly used for URL shorteners. Base conversion helps to convert the same number between its different number representation systems. Base 62 conversion is used as there are 62 possible characters for *hashValue*. Let us use an example to explain how the conversion works: convert $11157_{10}$ to base 62 representation ($11157_{10}$ represents 11157 in a base 10 system).

- From its name, base 62 is a way of using 62 characters for encoding. The mappings are: *0-0, ..., 9-9, 10-a, 11-b, ..., 35-z, 36-A, ..., 61-Z, where* 'a' stands for 10, 'Z' stands for 61, etc.
- $11157_{10} = 2 \times 62^2 + 55 \times 62^1 + 59 \times 62^0 = [2, 55, 59] \rightarrow [2, T, X]$ in base 62 representation. Figure 8-6 shows the conversation process.


Figure 8-6

• Thus, the short URL is https://tinyurl.com /**2TX**

**Comparison of the two approaches**

Table 8-3 shows the differences of the two approaches.

| **Hash + collision resolution** | **Base 62 conversion** |
|---|---|
| Fixed short URL length. | The short URL length is not fixed. It goes up with the ID. |
| It does not need a unique ID generator. | This option depends on a unique ID generator. |
| Collision is possible and must be resolved. | Collision is impossible because ID is unique. |
| It is impossible to figure out the next available short URL because it does not depend on ID. | It is easy to figure out the next available short URL if ID increments by 1 for a new entry. This can be a security concern. |

Table 8-3

## URL shortening deep dive

As one of the core pieces of the system, we want the URL shortening flow to be logically simple and functional. Base 62 conversion is used in our design. We build the following diagram (Figure 8-7) to demonstrate the flow.

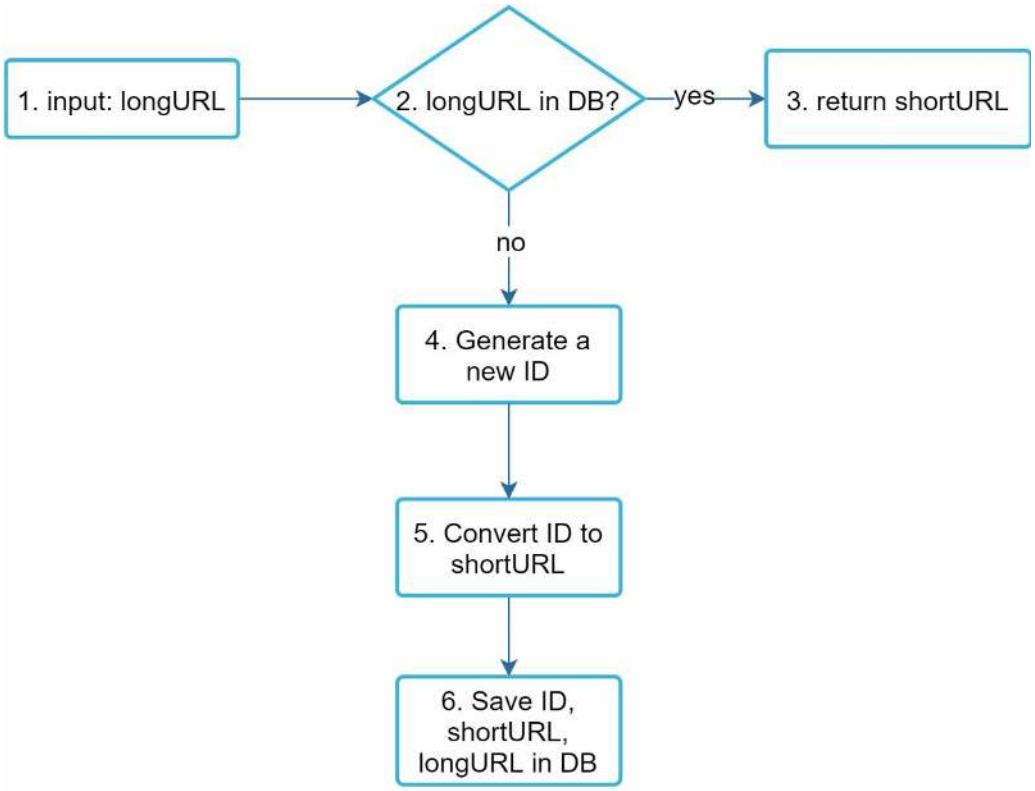


Figure 8-7

1. longURL is the input.
2. The system checks if the longURL is in the database.
3. If it is, it means the longURL was converted to shortURL before. In this case, fetch the shortURL from the database and return it to the client.
4. If not, the longURL is new. A new unique ID (primary key) Is generated by the unique ID generator.
5. Convert the ID to shortURL with base 62 conversion.
6. Create a new database row with the ID, shortURL, and longURL.

To make the flow easier to understand, let us look at a concrete example.

- Assuming the input longURL is: https://en.wikipedia.org/wiki/Systems_design
- Unique ID generator returns ID: 2009215674938.
- Convert the ID to shortURL using the base 62 conversion. ID (2009215674938) is converted to “zn9edcu”.
- Save ID, shortURL, and longURL to the database as shown in Table 8-4.

| id | shortURL | longURL |
|---|---|---|
| 2009215674938 | zn9edcu | https://en.wikipedia.org/wiki/Systems_design |

Table 8-4

The distributed unique ID generator is worth mentioning. Its primary function is to generate globally unique IDs, which are used for creating shortURLs. In a highly distributed

environment, implementing a unique ID generator is challenging. Luckily, we have already discussed a few solutions in “Chapter 7: Design A Unique ID Generator in Distributed Systems”. You can refer back to it to refresh your memory.

## URL redirecting deep dive

Figure 8-8 shows the detailed design of the URL redirecting. As there are more reads than writes, <*shortURL, longURL*> mapping is stored in a cache to improve performance.

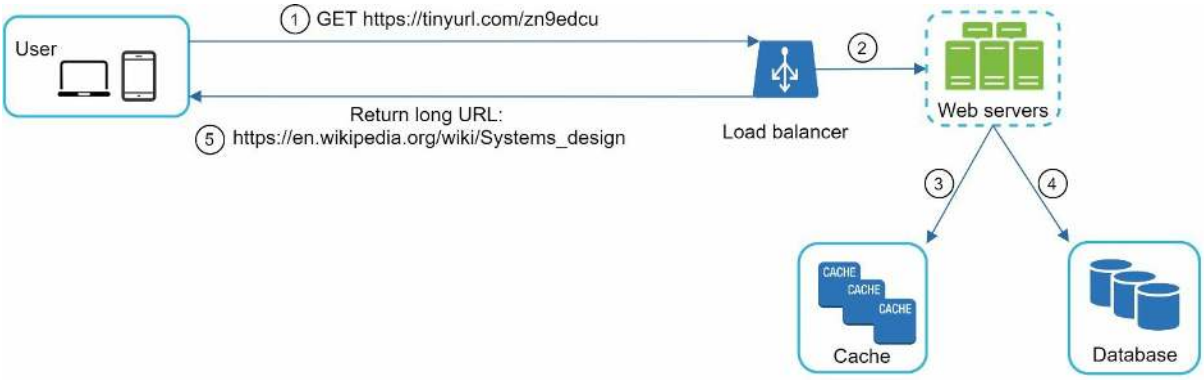


Figure 8-8

The flow of URL redirecting is summarized as follows:

1. A user clicks a short URL link: https://tinyurl.com/zn9edcu
2. The load balancer forwards the request to web servers.
3. If a shortURL is already in the cache, return the longURL directly.
4. If a shortURL is not in the cache, fetch the longURL from the database. If it is not in the database, it is likely a user entered an invalid shortURL.
5. The longURL is returned to the user.

## Step 4 - Wrap up

In this chapter, we talked about the API design, data model, hash function, URL shortening, and URL redirecting.

If there is extra time at the end of the interview, here are a few additional talking points.

- Rate limiter: A potential security problem we could face is that malicious users send an overwhelmingly large number of URL shortening requests. Rate limiter helps to filter out requests based on IP address or other filtering rules. If you want to refresh your memory about rate limiting, refer to “Chapter 4: Design a rate limiter”.
- Web server scaling: Since the web tier is stateless, it is easy to scale the web tier by adding or removing web servers.
- Database scaling: Database replication and sharding are common techniques.
- Analytics: Data is increasingly important for business success. Integrating an analytics solution to the URL shortener could help to answer important questions like how many people click on a link? When do they click the link? etc.
- Availability, consistency, and reliability. These concepts are at the core of any large system’s success. We discussed them in detail in Chapter 1, please refresh your memory on these topics.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

## Reference materials

[1] A RESTful Tutorial: https://www.restapitutorial.com/index.html

[2] Bloom filter: https://en.wikipedia.org/wiki/Bloom_filter

# CHAPTER 9: DESIGN A WEB CRAWLER

In this chapter, we focus on web crawler design: an interesting and classic system design interview question.

A web crawler is known as a robot or spider. It is widely used by search engines to discover new or updated content on the web. Content can be a web page, an image, a video, a PDF file, etc. A web crawler starts by collecting a few web pages and then follows links on those pages to collect new content. Figure 9-1 shows a visual example of the crawl process.

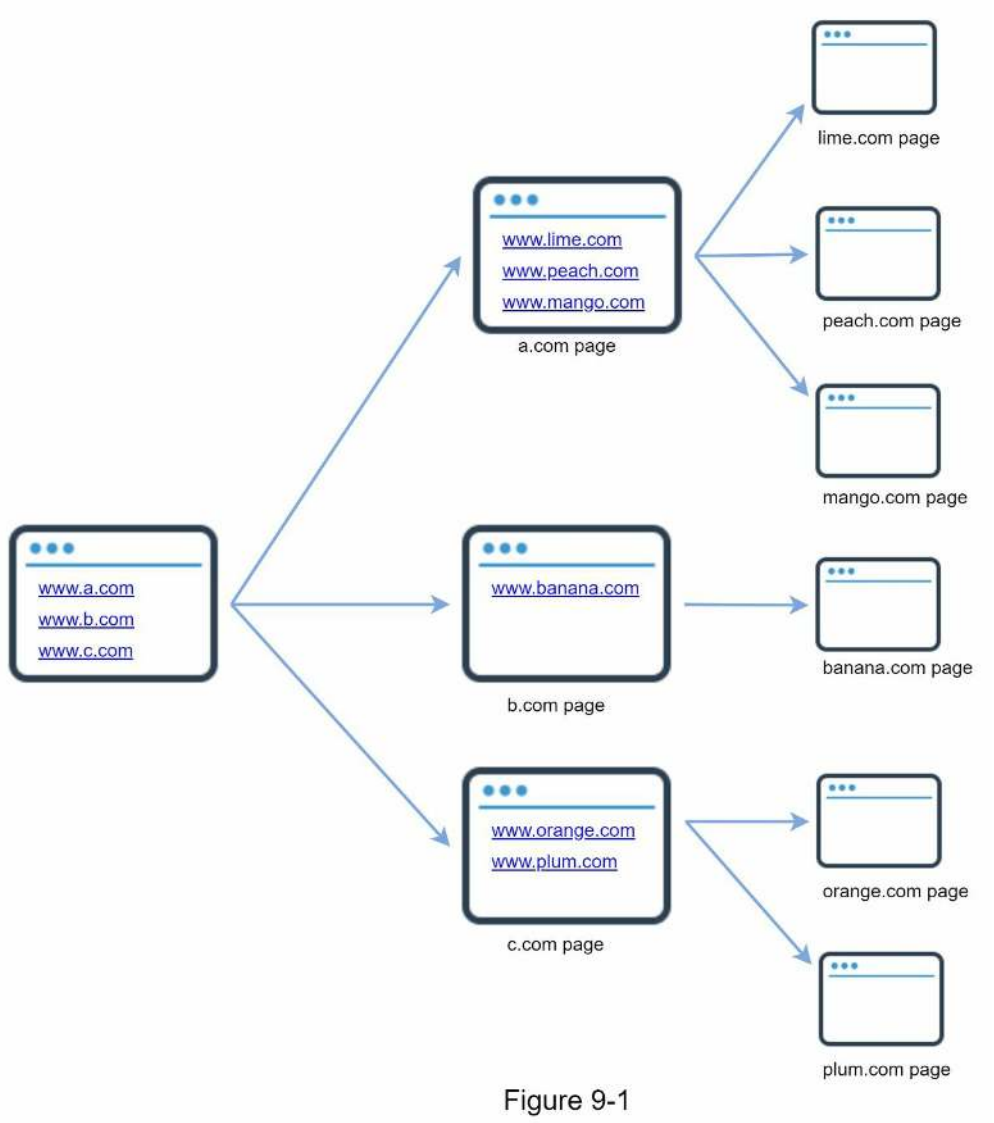


Figure 9-1

A crawler is used for many purposes:

- Search engine indexing: This is the most common use case. A crawler collects web pages to create a local index for search engines. For example, Googlebot is the web crawler behind the Google search engine.
- Web archiving: This is the process of collecting information from the web to preserve data for future uses. For instance, many national libraries run crawlers to archive web sites. Notable examples are the US Library of Congress [1] and the EU web archive [2].
- Web mining: The explosive growth of the web presents an unprecedented opportunity for

data mining. Web mining helps to discover useful knowledge from the internet. For example, top financial firms use crawlers to download shareholder meetings and annual reports to learn key company initiatives.

• Web monitoring. The crawlers help to monitor copyright and trademark infringements over the Internet. For example, Digimarc [3] utilizes crawlers to discover pirated works and reports.

The complexity of developing a web crawler depends on the scale we intend to support. It could be either a small school project, which takes only a few hours to complete or a gigantic project that requires continuous improvement from a dedicated engineering team. Thus, we will explore the scale and features to support below.

## Step 1 - Understand the problem and establish design scope

The basic algorithm of a web crawler is simple:

1. Given a set of URLs, download all the web pages addressed by the URLs.
2. Extract URLs from these web pages
3. Add new URLs to the list of URLs to be downloaded. Repeat these 3 steps.

Does a web crawler work truly as simple as this basic algorithm? Not exactly. Designing a vastly scalable web crawler is an extremely complex task. It is unlikely for anyone to design a massive web crawler within the interview duration. Before jumping into the design, we must ask questions to understand the requirements and establish design scope:

**Candidate**: What is the main purpose of the crawler? Is it used for search engine indexing, data mining, or something else?
**Interviewer**: Search engine indexing.

**Candidate**: How many web pages does the web crawler collect per month?
**Interviewer**: 1 billion pages.

**Candidate**: What content types are included? HTML only or other content types such as PDFs and images as well?
**Interviewer**: HTML only.

**Candidate**: Shall we consider newly added or edited web pages?
**Interviewer**: Yes, we should consider the newly added or edited web pages.

**Candidate**: Do we need to store HTML pages crawled from the web?
**Interviewer**: Yes, up to 5 years

**Candidate**: How do we handle web pages with duplicate content?
**Interviewer**: Pages with duplicate content should be ignored.

Above are some of the sample questions that you can ask your interviewer. It is important to understand the requirements and clarify ambiguities. Even if you are asked to design a straightforward product like a web crawler, you and your interviewer might not have the same assumptions.

Beside functionalities to clarify with your interviewer, it is also important to note down the following characteristics of a good web crawler:

• Scalability: The web is very large. There are billions of web pages out there. Web crawling should be extremely efficient using parallelization.

• Robustness: The web is full of traps. Bad HTML, unresponsive servers, crashes, malicious links, etc. are all common. The crawler must handle all those edge cases.

• Politeness: The crawler should not make too many requests to a website within a short time interval.

• Extensibility: The system is flexible so that minimal changes are needed to support new content types. For example, if we want to crawl image files in the future, we should not need to redesign the entire system.

## Back of the envelope estimation

The following estimations are based on many assumptions, and it is important to communicate with the interviewer to be on the same page.

• Assume 1 billion web pages are downloaded every month.

• QPS: 1,000,000,000 / 30 days / 24 hours / 3600 seconds = ~400 pages per second.

• Peak QPS = 2 * QPS = 800

• Assume the average web page size is 500k.

• 1-billion-page x 500k = 500 TB storage per month. If you are unclear about digital storage units, go through “Power of 2” section in Chapter 2 again.

• Assuming data are stored for five years, 500 TB * 12 months * 5 years = 30 PB. A 30 PB storage is needed to store five-year content.

## Step 2 - Propose high-level design and get buy-in

Once the requirements are clear, we move on to the high-level design. Inspired by previous studies on web crawling [4] [5], we propose a high-level design as shown in Figure 9-2.

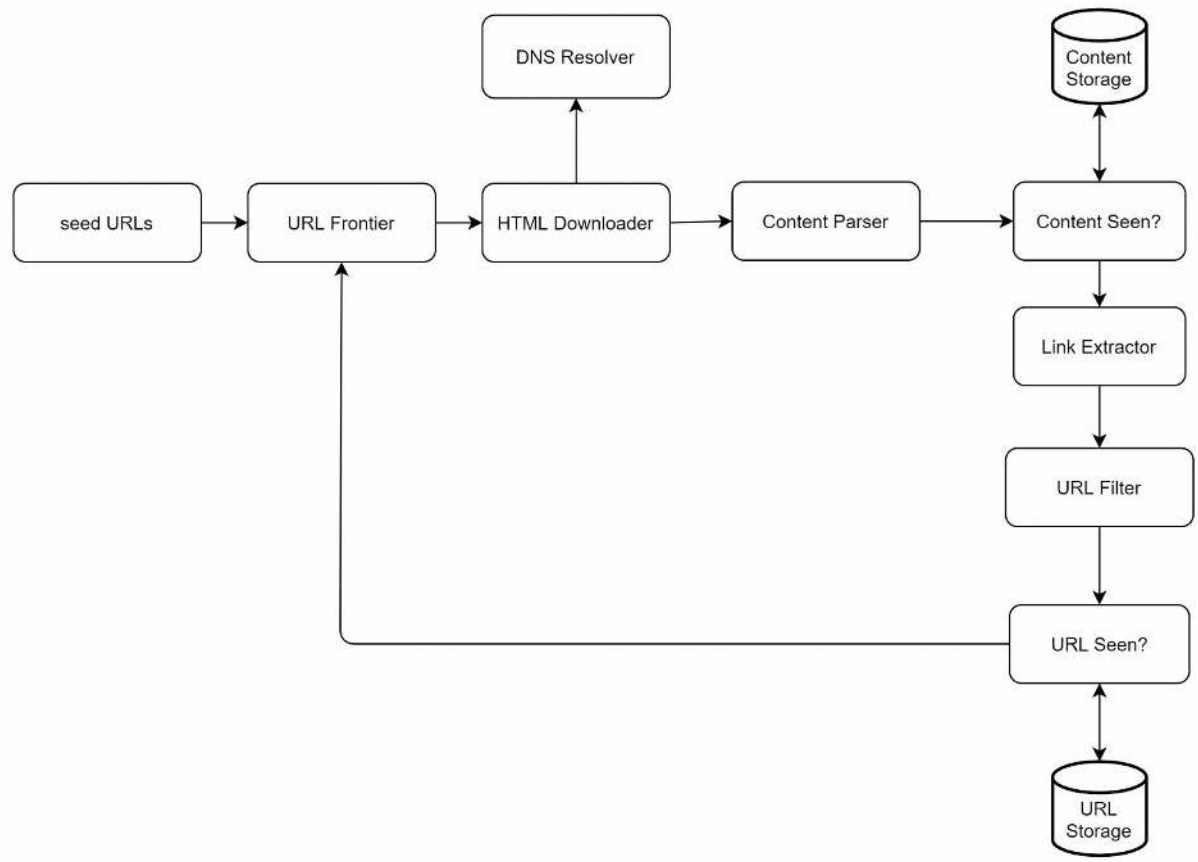


Figure 9-2

First, we explore each design component to understand their functionalities. Then, we examine the crawler workflow step-by-step.

### Seed URLs

A web crawler uses seed URLs as a starting point for the crawl process. For example, to crawl all web pages from a university's website, an intuitive way to select seed URLs is to use the university's domain name.

To crawl the entire web, we need to be creative in selecting seed URLs. A good seed URL serves as a good starting point that a crawler can utilize to traverse as many links as possible. The general strategy is to divide the entire URL space into smaller ones. The first proposed approach is based on locality as different countries may have different popular websites. Another way is to choose seed URLs based on topics; for example, we can divide URL space into shopping, sports, healthcare, etc. Seed URL selection is an open-ended question. You are not expected to give the perfect answer. Just think out loud.

### URL Frontier

Most modern web crawlers split the crawl state into two: to be downloaded and already downloaded. The component that stores URLs to be downloaded is called the URL Frontier. You can refer to this as a First-in-First-out (FIFO) queue. For detailed information about the URL Frontier, refer to the deep dive.

**HTML Downloader**

The HTML downloader downloads web pages from the internet. Those URLs are provided by the URL Frontier.

**DNS Resolver**

To download a web page, a URL must be translated into an IP address. The HTML Downloader calls the DNS Resolver to get the corresponding IP address for the URL. For instance, URL www.wikipedia.org is converted to IP address 198.35.26.96 as of 3/5/2019.

**Content Parser**

After a web page is downloaded, it must be parsed and validated because malformed web pages could provoke problems and waste storage space. Implementing a content parser in a crawl server will slow down the crawling process. Thus, the content parser is a separate component.

**Content Seen?**

Online research [6] reveals that 29% of the web pages are duplicated contents, which may cause the same content to be stored multiple times. We introduce the "Content Seen?" data structure to eliminate data redundancy and shorten processing time. It helps to detect new content previously stored in the system. To compare two HTML documents, we can compare them character by character. However, this method is slow and time-consuming, especially when billions of web pages are involved. An efficient way to accomplish this task is to compare the hash values of the two web pages [7].

**Content Storage**

It is a storage system for storing HTML content. The choice of storage system depends on factors such as data type, data size, access frequency, life span, etc. Both disk and memory are used.

- Most of the content is stored on disk because the data set is too big to fit in memory.
- Popular content is kept in memory to reduce latency.

**URL Extractor**

URL Extractor parses and extracts links from HTML pages. Figure 9-3 shows an example of a link extraction process. Relative paths are converted to absolute URLs by adding the "https://en.wikipedia.org" prefix.

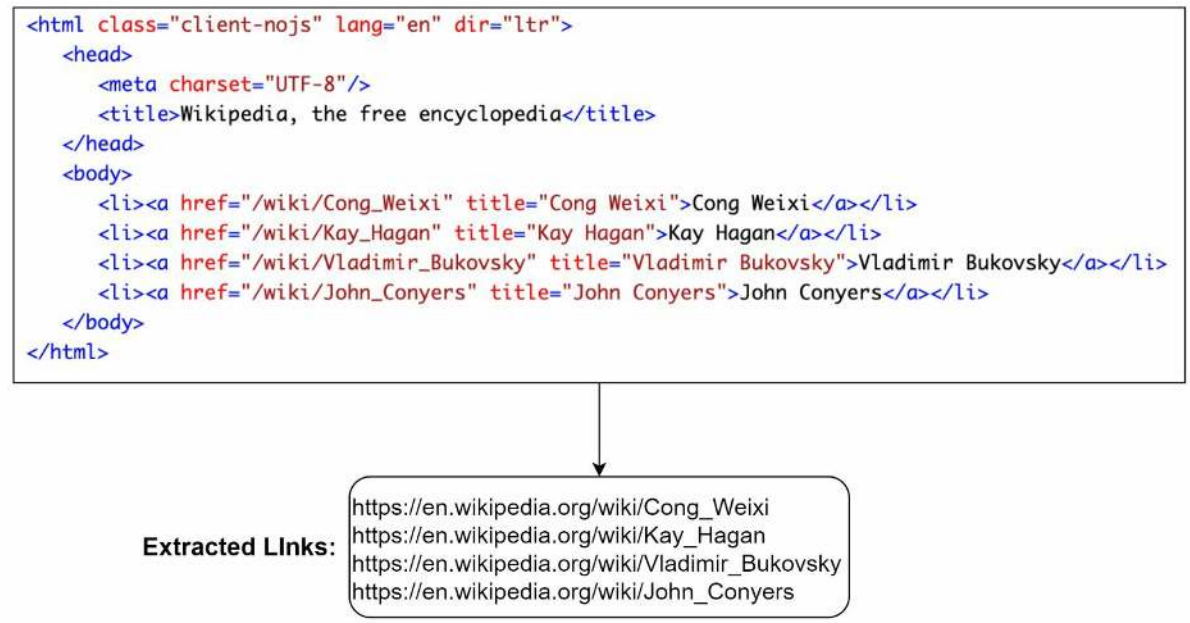


Figure 9-3

### URL Filter

The URL filter excludes certain content types, file extensions, error links and URLs in "blacklisted" sites.

### URL Seen?

"URL Seen?" is a data structure that keeps track of URLs that are visited before or already in the Frontier. "URL Seen?" helps to avoid adding the same URL multiple times as this can increase server load and cause potential infinite loops.

Bloom filter and hash table are common techniques to implement the "URL Seen?" component. We will not cover the detailed implementation of the bloom filter and hash table here. For more information, refer to the reference materials [4] [8].

### URL Storage

URL Storage stores already visited URLs.

So far, we have discussed every system component. Next, we put them together to explain the workflow.

### Web crawler workflow

To better explain the workflow step-by-step, sequence numbers are added in the design diagram as shown in Figure 9-4.

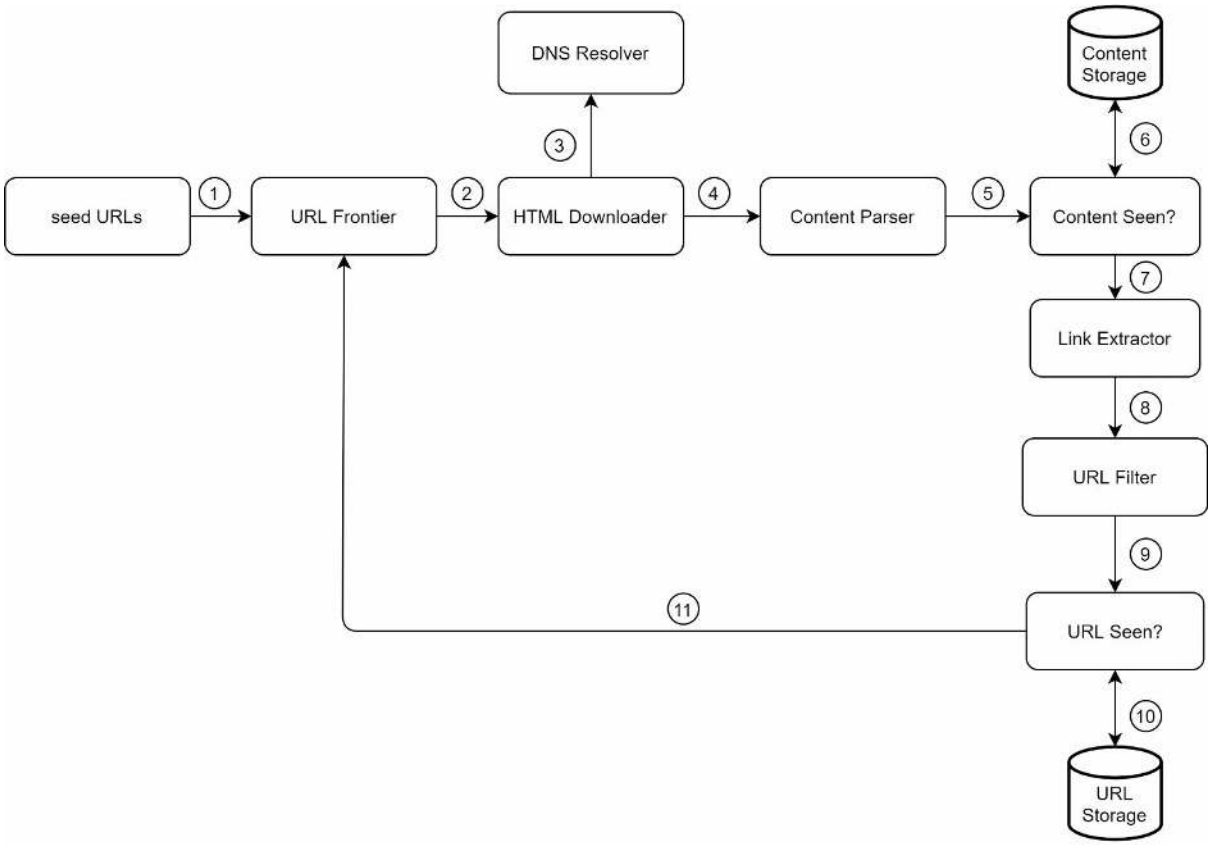


Figure 9-4

Step 1: Add seed URLs to the URL Frontier

Step 2: HTML Downloader fetches a list of URLs from URL Frontier.

Step 3: HTML Downloader gets IP addresses of URLs from DNS resolver and starts downloading.

Step 4: Content Parser parses HTML pages and checks if pages are malformed.

Step 5: After content is parsed and validated, it is passed to the "Content Seen?" component.

Step 6: "Content Seen" component checks if a HTML page is already in the storage.

- If it is in the storage, this means the same content in a different URL has already been processed. In this case, the HTML page is discarded.
- If it is not in the storage, the system has not processed the same content before. The content is passed to Link Extractor.

Step 7: Link extractor extracts links from HTML pages.

Step 8: Extracted links are passed to the URL filter.

Step 9: After links are filtered, they are passed to the "URL Seen?" component.

Step 10: "URL Seen" component checks if a URL is already in the storage, if yes, it is processed before, and nothing needs to be done.

Step 11: If a URL has not been processed before, it is added to the URL Frontier.

## Step 3 - Design deep dive

Up until now, we have discussed the high-level design. Next, we will discuss the most important building components and techniques in depth:

- Depth-first search (DFS) vs Breadth-first search (BFS)
- URL frontier
- HTML Downloader
- Robustness
- Extensibility
- Detect and avoid problematic content

## DFS vs BFS

You can think of the web as a directed graph where web pages serve as nodes and hyperlinks (URLs) as edges. The crawl process can be seen as traversing a directed graph from one web page to others. Two common graph traversal algorithms are DFS and BFS. However, DFS is usually not a good choice because the depth of DFS can be very deep.

BFS is commonly used by web crawlers and is implemented by a first-in-first-out (FIFO) queue. In a FIFO queue, URLs are dequeued in the order they are enqueued. However, this implementation has two problems:

- Most links from the same web page are linked back to the same host. In Figure 9-5, all the links in wikipedia.com are internal links, making the crawler busy processing URLs from the same host (wikipedia.com). When the crawler tries to download web pages in parallel, Wikipedia servers will be flooded with requests. This is considered as “impolite”.

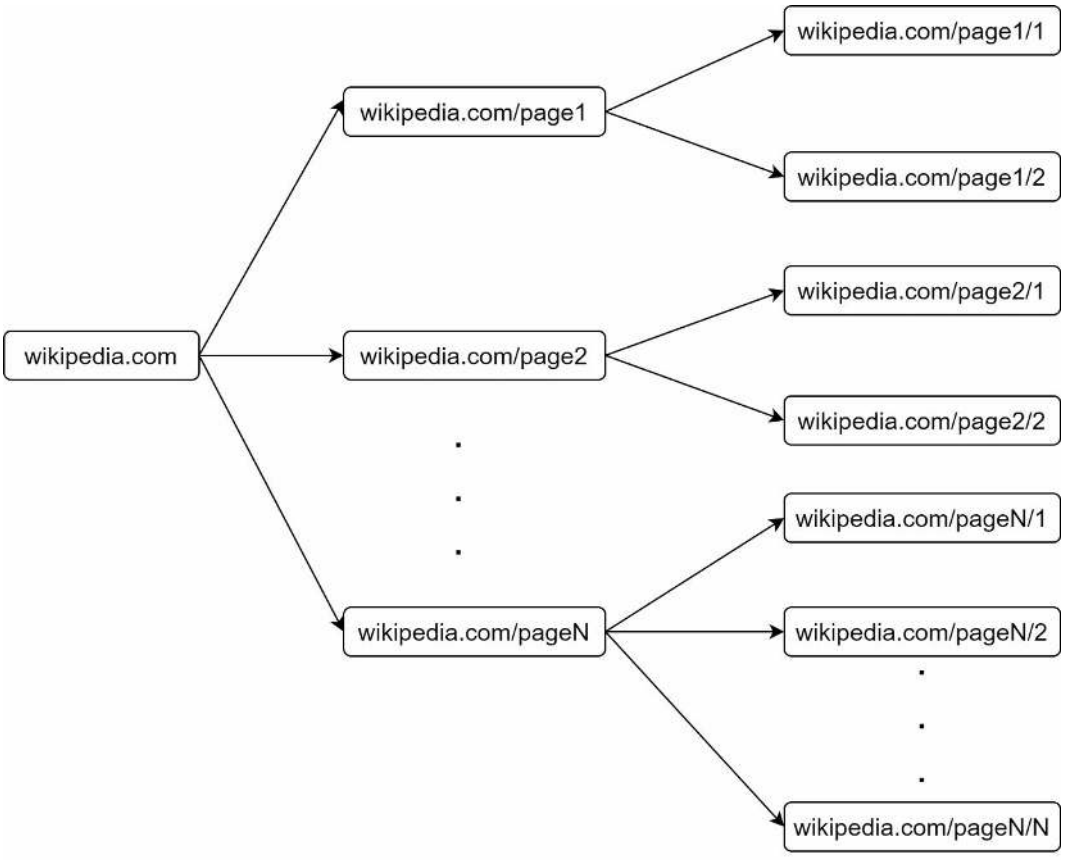


Figure 9-5