**DESIGN A Web Crawler**

A web crawler is known as a spider or robot. It is widely used by search engines to discover new or updated content on the web. A web crawler starts by collecting a few web pages and then follows links on those pages to collect new content.

A crawler is used for many purposes:  
1. Search engine indexing: A crawler collects web pages to create a local index for search engines.

2. Web archiving: Process of collecting information from the web to preserve data for future use. For instance, many national libraries run crawlers to archive websites. Example: US Library of Congress and EU web archive.

3. Web 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.

4. 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.

### Understanding Problem and Establishing 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.

Query1: Which type of Content will the web crawler be consuming?

-> HTML only.

Query2: How many web pages does the web crawler collect per month?

-> 1 billion pages

Query3: What is the main purpose of the web crawler?

-> Search Engine indexing

Query4: Shall we consider newly added or edited web pages?

-> Yes, we should consider the newly added or edited web pages.  
Query5: Do we need to store HTML pages crawled from the web?

-> Yes, up to 5 years.  
Query6: How do we handle web pages with duplicate content?

-> Pages with duplicate content should be ignored.

Characteristics of a good web crawler:

1. Scalability: The web is very large. There are billions of web pages out there. Web crawling should be extremely efficient using parallelization.
2. 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.
3. Politeness: The crawler should not make too many requests to a website within a short time interval.
4. 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 design the entire system.

### Back of the Envelope Estimation:

1 billion web pages are downloaded every month

QPS : 1000,000,000 / 30days / 24hours/ 3600 seconds ~ 1,000,000,000/2392000 ~ 400 pages per sec

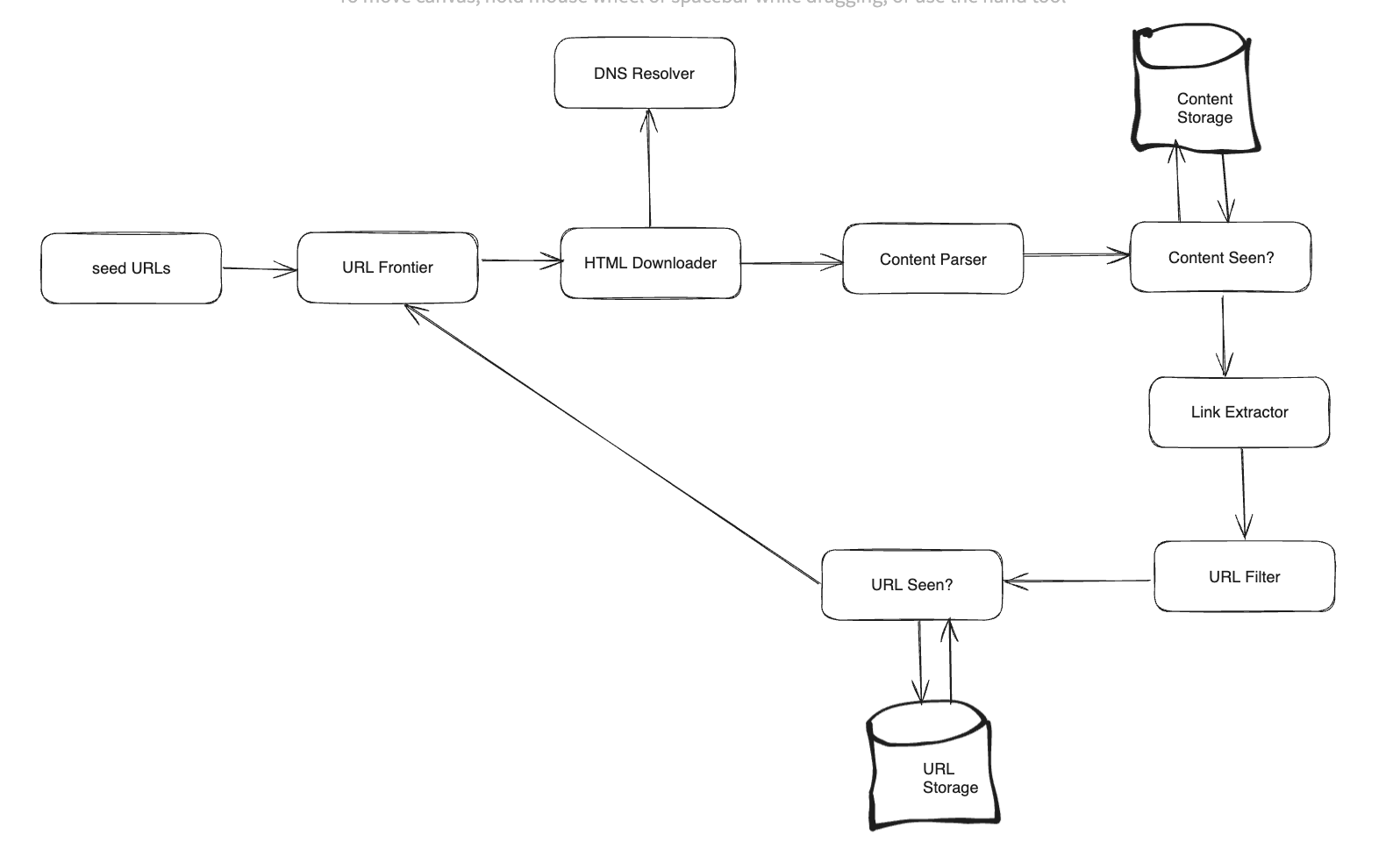
Peak QPS 800 web pages per sec

Assume average page size as 500KB

So 1billion page \* 500KB = 5\*10^11 KB = 500 TB

Assuming data to be stored for 5 years, 500TB\*12 months\* 5 = 30000TB = 30 PB.

### High Level Design Propositions and approaches:



**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.  
A good seed URLserves 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 small ones.   
The first 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.

**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**. This can be a FIFO queue.

**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.

**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?**We introduce “Content Seen?” data structure to eliminate data redundancy and shorten processing time. It helps to detect new content previously stored by character.  
An efficient way to accomplish this task is to compare the hash values of the two web pages.

**Content Storage**

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.

**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.

**URL storage**

URL storage store already visited URLs

### Design Deep Dive:

**DFS vs BFS:**You can think of the web as a directed graph where web pages serve as nodes and hyperlinks(URLs) as edges. **DFS is usually not a good choice because depth of DFS can be very deep.**

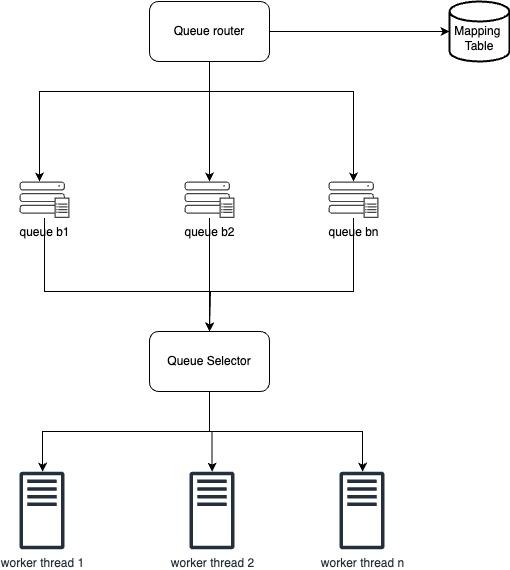
**BFS is commonly used by web crawlers and is implemented by a FIFO queue.**However this implementation has two problems:

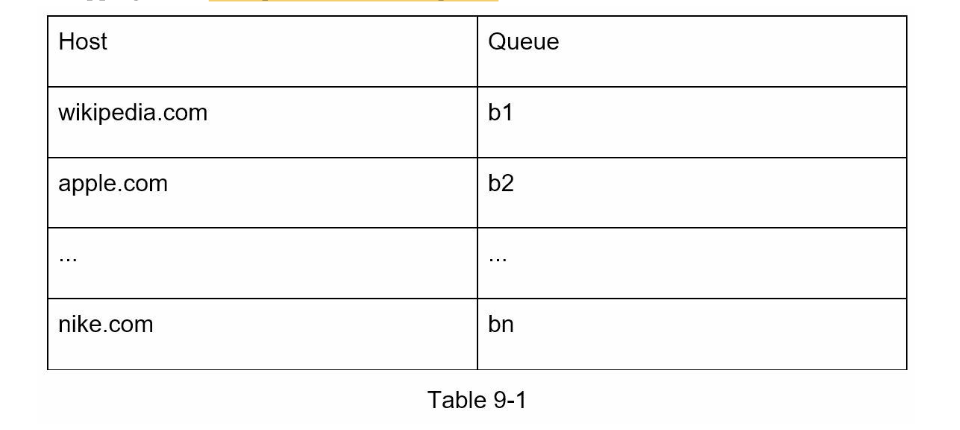
1. Most links from the same web page are linked back to the same host.**When the crawler tries to download web pages in parallel, the host servers will be flooded with requests. This is considered as impolite.**
2. Standard BFS does not take the priority of a URL into consideration. **The web is large and not every page has the same level of quality and importance. Therefore, we may want to prioritize URLs according to their page ranks, web traffic, update frequency,etc.**

**URL frontier**

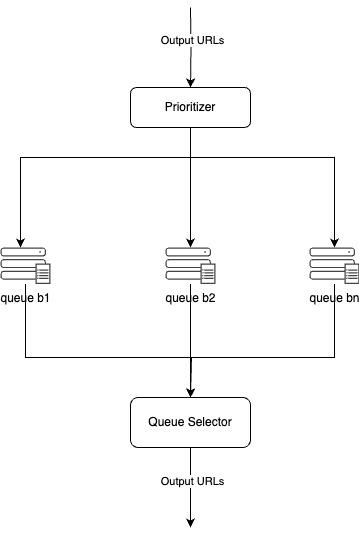
The URL frontier is an important component to ensure politeness, URL prioritization and freshness.  
**Politeness:**Generally, a web crawler should avoid sending too many requests to the same hosting server within a short period. Sending too many requests is considered as “impolite” or even treated as a denial-of-service (DOS) attack.

The general idea of enforcing politeness is to download one page at a time from the same host. A delay can be added between two download tasks. The politeness constraint is implemented by maintaining a mapping from website hostnames to download(worker) threads.



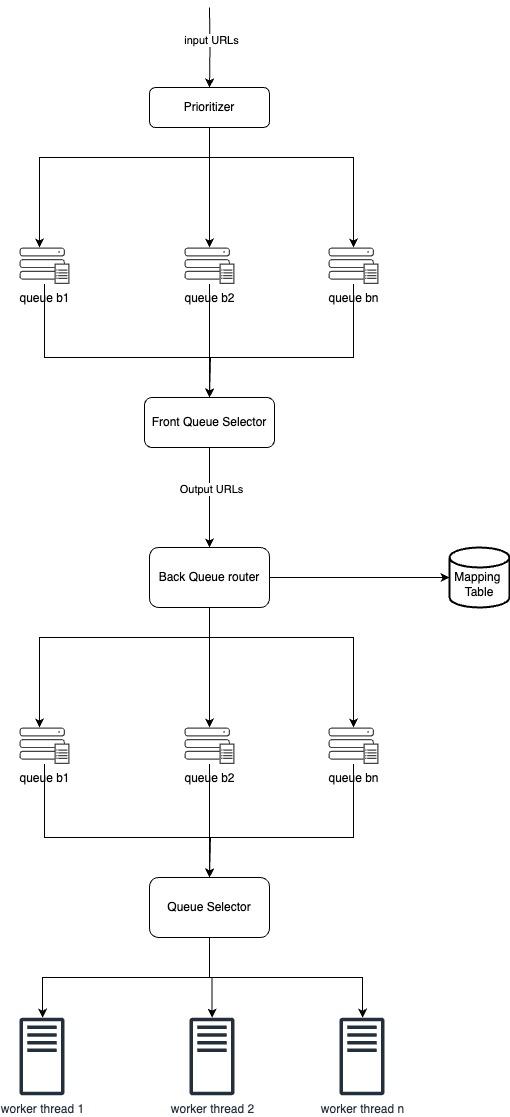
* Queue router: It ensures that each queue (b1, b2, .. bn) only contains URLs from the same host
* Mapping table: It maps each host to a queue
* FIFO queues b1, b2 to bn: Each queue contains URLs from the same host
* Queue selector: Each worker thread is mapped to a FIFO queue, and it only downloads URLs from that queue. The queue selection logic is done by the Queue selector.
* Worker thread 1 to N: A worker thread downloads web pages one by one from the same host. A delay can be added between two download tasks.

**Priority:**

We prioritize URLs based on usefulness, which can be measured by PageRank, website traffic, update frequency, etc. “Prioritizer” is the component that handles URL prioritization.

* Prioritizer: It takes URLs as input and computes the priorities.
* Queue f1 to fn: Each queue has an assigned priority. Queues with high priority are selected with higher probability.
* Queue selector: Randomly chooses a queue with a bias towards queues with higher priority.

Below figure represent the URL frontier design and it contains two modules:

1. Front queues: manage prioritization
2. Back queues: manage politeness

**Freshness**

Web pages are constantly being added, deleted and edited. A web crawler must periodically recrawl downloaded pages to keep our data set fresh.

Few strategies to optimize freshness are listed as follows:

* Recrawl based on web pages update history.
* PrioritizeURLs and recrawl important pages first and more frequently.

**Storage for URL frontier**

In real world crawl for search engines, the number of URLs in the frontier could be hundreds of millions. Putting everything in memory is undesirable either because the disk is slow and it can easily become a bottleneck for the crawl.

We adopted a hybrid approach. The majority of URLs are stored on disk, so the storage space is not a problem. To reduce the cost of reading from the disk and writing to disk, we maintain buffers in memory for dequeue/enqueue operations. Data in the buffer is periodically written to the disk.

**HTML Downloader**

* **Rots.txt**(Robots Exclusion Protocol) is a standard used by websites to communicate with crawlers. It specifies what pages crawlers are allowed to download. Before attempting to crawl a web site, a crawler should check its corresponding robots.txt first and follow its rules.

To avoid repeat downloads of robots.txt file we cache the results of the file. The file is downloaded and saved to cache periodically.

Here is a piece of robots.txt file taken from https://www.amazon.com/robots.txt. Some of the directories like creatorhub are disallowed for Google bot.

User-agent: Googlebot

Disallow: /creatorhub/\*

Disallow: /rss/people/\*/reviews

Disallow: /gp/pdp/rss/\*/reviews

Disallow: /gp/cdp/member-reviews/

Disallow: /gp/aw/cr/

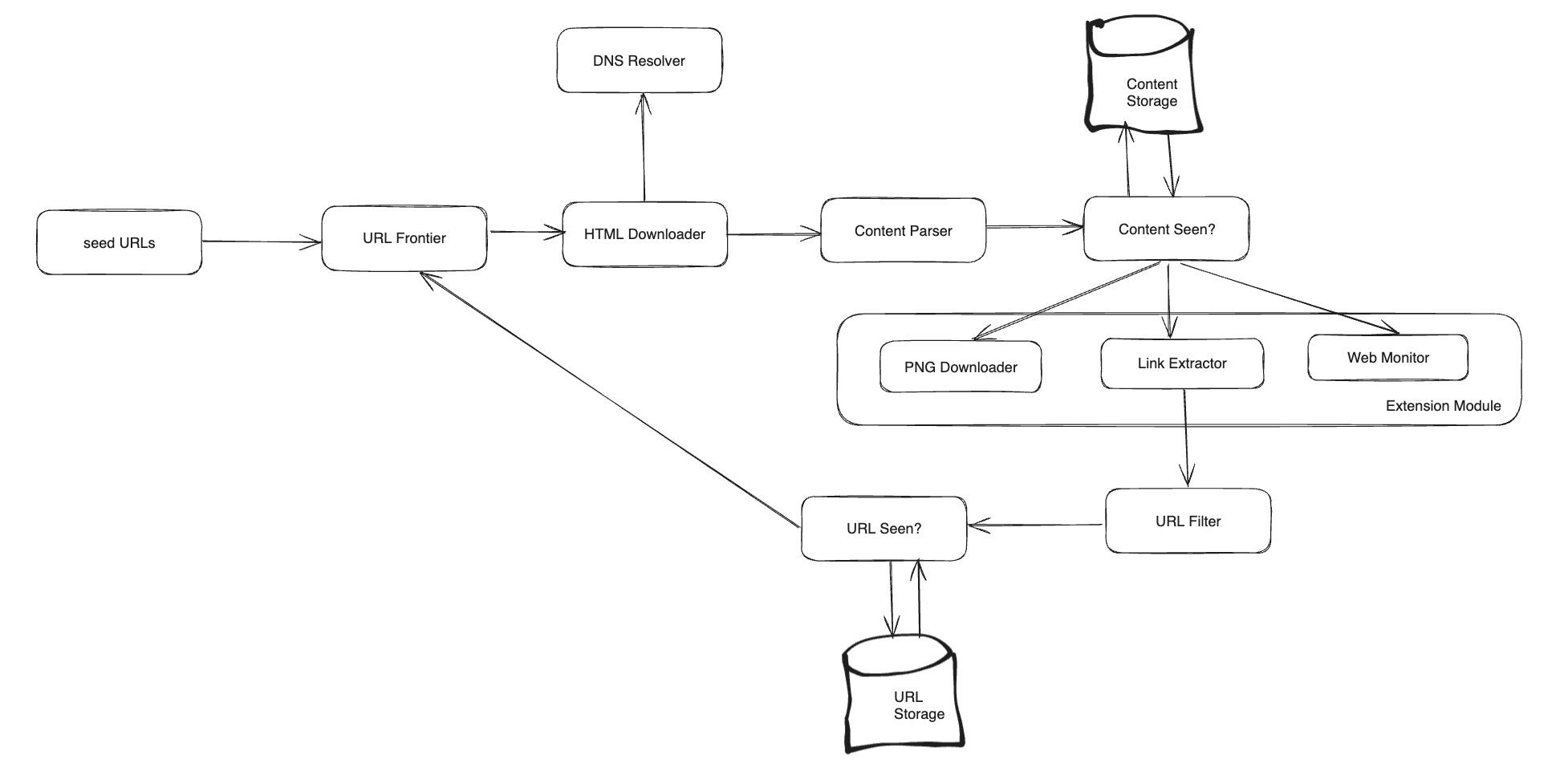
* **Performance optimization**

1. **Distributed crawl:** To achieve high performance, crawl jobs are distributed into multiple servers, and each server runs multiple threads. The URL space is partitioned into smaller pieces, so each downloader is responsible for a subset of the URLs.
2. **Cache DNS Resolver:** DNS Resolver is a bottleneck for crawlers because DNS requests might take time due to the synchronous nature of many DNS interfaces. DNS response time ranges from 10ms to 200ms. Once a DNS is carried out by a crawler thread, other threads are blocked until the first request is completed. Maintaining our DNS cache to avoid calling DNS frequently is an effective technique for speed optimization. Our DNS cache keeps the domain name to IP address mapping and is updated periodically by cron jobs.
3. **Locality:**Distribute crawl servers geographically. When crawl servers are closer to website hosts, crawlers experience faster download time.
4. **Short timeout:**  
   To avoid long wait time, a maximal wait time is specified. If a host does not respond within a predefined time, the crawler will stop the job and crawl some other pages.

* **Robustness**

1. Consistent Hashing: Will help to distribute loads among downloaders. A new downloader server can be added or removed using consistent hashing.
2. Save crawl states and data: To guard against failures, crawl states and data are written to a storage system. A disrupted crawl can be restarted easily by loading saved states and data.
3. Exception handling: Errors are inevitable and common in a large-scale system. The crawler must handle exceptions gracefully without crashing the system.
4. Data validation: This is an important measure to prevent system errors.

* **Extensibility**

****

* **Detect and avoid problematic content**

1. Redundant content: Hashes or checksums help to detect duplications.
2. Spider traps:  
   For an infinite deep directory as listed like: *www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/...*

Such spider traps can be avoided by setting a maximal length for URLs. It is hard to develop automatic algorithms to avoid spider traps; however, a user can manually

1. Data Noise: Some of the contents have little or no value, such as advertisements, code snippets, spam URLs, etc. Those contents are not useful for crawlers and should be excluded if possible.

### **Additional optimization:**

Server-side rendering: Numerous websites use scripts like JavaScript, AJAX, etc to generate links on the fly. If we download and parse web pages directly, we will not be able to retrieve dynamically generated links. To solve this problem, we perform server-side rendering (also called dynamic rendering) first before parsing a page [12].

• Filter out unwanted pages: With finite storage capacity and crawl resources, an anti-spam component is beneficial in filtering out low quality and spam pages [13] [14].

• Database replication and sharding: Techniques like replication and sharding are used to improve the data layer availability, scalability, and reliability.

• Horizontal scaling: For large scale crawl, hundreds or even thousands of servers are needed to perform download tasks. The key is to keep servers stateless.

• Availability, consistency, and reliability: These concepts are at the core of any large system’s success. We discussed these concepts in detail in Chapter 1. Refresh your memory on these topics.

• Analytics: Collecting and analyzing data are important parts of any system because data is key ingredient for fine-tuning.