# Design a Web crawler



## Description

A web crawler (discover new and updated content on the web) is used for the following major purposes :

1. Search Engine indexing : create inverted index for search
2. Archiving of Web content : preserving data from the web for future use
3. Mining the Web for useful information : extract information from the web
4. Monitoring the web for copyright or trademarks infringement.

This design will be covering the Search Engine indexing use case.

# Crawler Concept

A simple web crawler problem description can be found here :
https://leetcode.com/problems/web-crawler/ (leetcode 1236)

The above question gives a flavor for a basic web-crawler algorithm. Although the above question restricts the search to the startUrl hostname, the summary of the overarching algorithm is :

1. Start from the page: `startUrl`
2. Call `HtmlParser.getUrls(url)` to get all urls from a webpage of given url.
3. Do not crawl the same link twice.
4. Add new URLs to the list

As can be seen from the above question, with billions of web pages, parallel processing of the web pages is a must. This extends to multi-threaded web crawler covered here

https://leetcode.com/problems/web-crawler-multithreaded/ (leetcode 1242)

In addition, a good web-crawler needs to satisfy additional requirements :

1. **Politeness**. It means that the crawler should not be sending too many requests to the hosting server within a short interval . Sometimes this can also lead to **'Denial-Of-Service'** with the web server denying requests. Too many crawl requests is considered impolite in the crawling world. Instead of searching all the related URLs on a single URL host at the same time, a polite crawler will either stagger or add delays to the download requests.
2. **Robots.txt**: (called Robots Exclusion Protocol) This is provided by the administrator of the hosting server. Robots.txt details as to what information can be retrieved from a specific server and information that is deemed restricted or unauthorized. Honoring the **'robots.txt'** is thus a requirement. A sample Robots.txt is below :

```
User-agent: Googlebot
Disallow: /creatorhub/*
Disallow: /rss/people/*/reviews
Disallow: /gp/pdp/rss/*/reviews
Disallow: /gp/cdp/member-reviews/
Disallow: /gp/aw/cr/
```

3. **Latest content** : Whenever the crawler retrieves the URL based information, that information needs to be **fresh**. What it means is that stale information cannot be

published or refresh rate cannot be too frequent, affecting the **politeness** policy. The design needs to be adaptable.

4. **Robustness** : The crawler should be able to detect nested deep URLs, which could lead to circular or infinite loops. It should be able to detect **spam** and / or **malicious** content ,; handle unresponsive servers, server crashes.
5. **Extensible** : New content types often get added and hence supporting additional content types should not need a design overhaul.
6. Most crawler jobs are initiated by stateless workers running on servers/spot-instances on the cloud. There is a cost associated with such instances and minimizing the cost is a goal for crawling jobs.

The **seed URL** is the base URL that will initiate the crawling process. Based on the above, the functional requirements can be outlined as below:

## Functional Requirement

1. Create search inverted index using web crawler
2. Crawl only for HTML content (PDF, Image, Video are not part of the functional requirement)
3. Crawler can update indices of updated pages
4. Crawler can update indices with new pages
5. Crawler must avoid duplicate content, and non-safety content

## Non Functional Requirement

1. HTML page content to be relevant for 10 years
2. Design should avoid bursts of requests to the hosting server
    a. (This will entail resource usage on the hosting server and hence needs to have a politeness quotient!)
3. Spam and obnoxious content should be identified and ignored as part of the crawler.

## Back of Envelope Calculations

**Storage Needs:**

1. Assume that 2 Billion web pages are downloaded every month
2. Queries Per second = 2 * 10 ^9 / 30 days / 24 hours / 3600 seconds = 800 pages / second
3. Assume each page 500KB, so 2 Billion pages = 1PB storage per month, 12PB per year
4. 120PB for 10 year storage

**Latency / Response Needs:**

1. The crawled web page should be displayed reasonably within 1 second, based upon the type of content and connectivity needs specified.

# Single Node Design

## Leetcode 1236 (Single Node Web Crawler)

https://leetcode.com/problems/web-crawler/ (leetcode 1236)

The web can be thought of as a directed graph where

1. web page serve as nodes and
2. hyperlinks (URLs) as edges

The crawl process can be thought of as traversing a directed graph from one page to another. So, traversal could be either DFS or BFS based. With billions of interconnected web pages, DFS can have very long queues. BFS is usually the choice for web crawler traversal.

```
class Solution {
public:
  std::string gethostname(std::string url)
  {
    url = url.substr(url.find("//") + 2);
    return url.substr(0, url.find('/'));
  }

  // Web crawler using standard BFS
  std::vector<std::string> crawl(std::string startUrl, HtmlParser htmlParser) {
    // This is kept so that we dont crawl the same link twice.
    std::unordered_set<std::string> unique_urls;

    // For BFS search.
    std::queue<std::string> q;

    // Get the hostname.
    std::string hostname = gethostname(startUrl);
    q.push(startUrl);
    unique_urls.insert(startUrl);

    // Start BFS.
    while(!q.empty())
    {
      int level_size = q.size();
      // level order traversal
      for (int i=0; i < level_size; ++i) {
        string urlcrawl = q.front();
```

```
            q.pop();

            for(auto url : htmlParser.getUrls(urlcrawl))
            {
                if(url.find(hostname) != string::npos &&
                    !unique_urls.count(url))
                {
                   q.push(url);
                   unique_urls.insert(url);
                }
            }
        }
    }

    return (std::vector<string>(unique_urls.begin(), unique_urls.end()));
  }
};
```

## Leetcode 1242 (Multi-threaded Web Crawler)

Shows a multithreaded solution with multiple worker threads crawling.

```
class Solution {
public:
  vector<string> crawl(string startUrl, HtmlParser htmlParser) {
    parser_ = &htmlParser;
    visit_hostname_ = Hostname(startUrl);
    urls_.clear();
    seen_.clear();

    urls_.push_back(startUrl);
    seen_.insert(startUrl);

    std::vector<std::thread> workers;
    // 2 workers crawling showing the coordination needed
    for (int i = 0; i < 2; ++i) {
      workers.push_back(std::thread(&Solution::Crawl, this));
    }
    for (auto& t : workers) {
      t.join();
    }

    return std::move(urls_);
  }

private:
  static std::string_view Hostname(std::string_view url) {
```

```
  const int start = url.find("://") + 3;
  const int end = url.find('/', start);
  if (url.npos == end) {
    return url.substr(start);
  }
  return url.substr(start, end - start);
}

void Crawl() {
  while (true) {
    std::string url;
    {
      std::unique_lock<std::mutex> lock(mtx_);
      if (urls_.size() == next_to_visit_) {
        // Do not terminate earlier if another worker can stil generate additional urls.
        cond_.wait(lock, [this]() {
          return (next_to_visit_ < urls_.size()) || (num_crawling_ == 0);
        });
        if (urls_.size() == next_to_visit_) {
          return;
        }
      }
      url = urls_[next_to_visit_++];
      ++num_crawling_;
    }

    std::vector<std::string> outlinks = parser_->getUrls(url);

    {
      std::lock_guard<std::mutex> lock(mtx_);
      --num_crawling_;
      for (std::string& link : outlinks) {
        if (Hostname(link) != visit_hostname_) continue;
        if (seen_.count(link)) continue;
        urls_.push_back(link);
        seen_.insert(std::move(link));
      }
    }
    cond_.notify_all();
  }
}

std::mutex mtx_;
std::condition_variable cond_;
HtmlParser* parser_;
std::string_view visit_hostname_;
std::vector<std::string> urls_;
int next_to_visit_ = 0;
std::unordered_set<std::string> seen_;
```

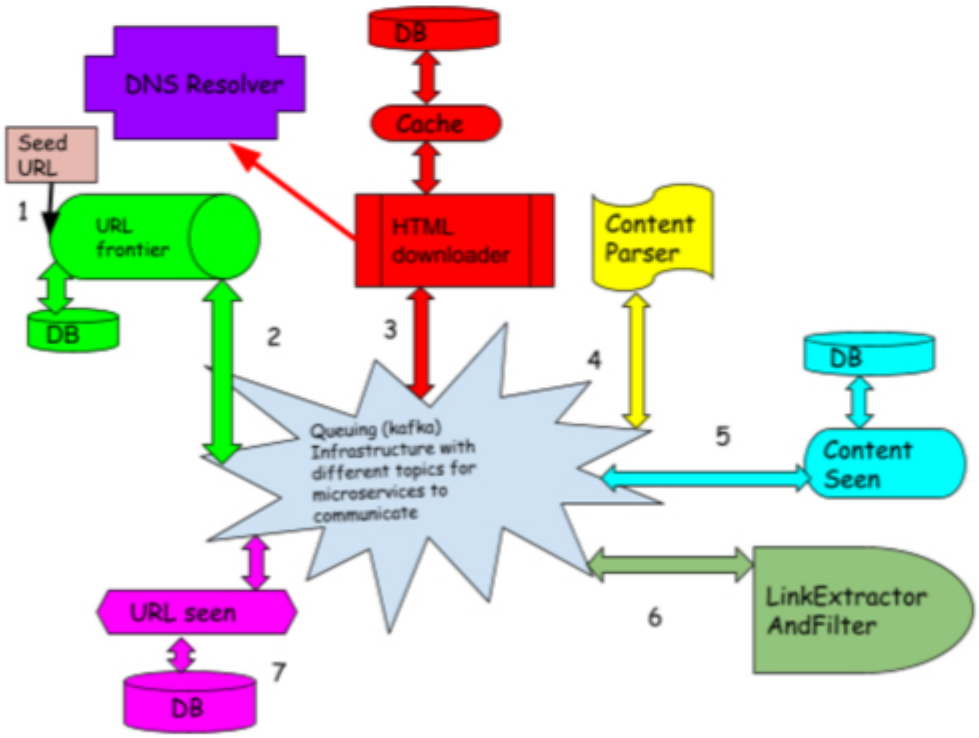
```
    int num_crawling_ = 0;

};
```

## Functional Design

As can be seen from the above two leetcode solutions we need the following

1. **Seed URLs** : starting point for the crawl process
2. **URL frontier** : URLs that are to be downloaded
3. **DNS resolver** : get the IP address corresponding to an URL
4. **Content Parser** : Parse and validate malformed web pages
5. **Content Seen** : 29% of web-pages are duplicated contents as per online research. To eliminate data redundancy and processing time, it helps to detect new content previously stored in the system : use hash values on web-pages content)
6. **Content Storage**: storage to store the HTML content
    a. Memory : store popular web pages content
    b. Most other content is stored in memory since the storage is too huge
7. **URL Extractor** :
    a. Implements getURLs() above
    b. Converts relative URLs to absolute paths (normalization)
8. **URL filter** : removes blacklisted URLs, file extensions, error links
9. **URL seen** : already visited URLs excluded using either bloom filter or hash tables
10. **URL storage** : stores the already visited URLs



The above figure shows the different microservices and how they communicate with each other.

1. Add **seed URLs** to the **URL frontier** :
    a. selection of good seed URLs could be based on country, topics (sports, shopping, healthcare);
    b. a good seed URL allows to traverse maximum links; URL frontier microservice decides which URL should be crawled next.
2. **URL frontier**
    a. decides which is the next URL to be crawled
    b. It keeps a URL priority table for the selection mechanism
3. **HTML downloader** (stateless worker) gets list of URLs from URL frontier
    a. HTML downloader gets IP address of URLs from **DNS resolvers** and downloads the web page, we need DNS resolvers because
        i. DNS servers do not support too many parallel DNS resolutions so the URL to IP address resolution can be cached. Response times range from 5ms to ~100ms.
        ii. Native library for DNS resolution is single-threaded causing higher level parallelism blocked in dns resolution
    b. Each worker saves the crawl state and web content into a DB
    c. The content and state are also saved into a distributed cache (popular pages)
4. **Content parser**
    a. parses the web content and
    b. checks malformed web pages
5. **Content seen** checks the web contents against the stored web content in **Content Storage** and only if not present then passes the web content to **LinkExtractorAndFilter**
    a. The check can use a hash of the web content stored in a key-value store
6. **LinkExtractorAndFilter** extracts the links from the web content and passes them through the Compute Intensive URL filter rules .
7. **URL Seen** takes the input URLs after the filter and if the URLs are not traversed earlier or meets an update/TTL criteria (saved in **URL storage**) adds them to the URL frontier.

## Detailed Design of Microservices

Implementing a Content parser on a Crawl server will slow down the Crawling process. Similarly failure in HTML downloading should not bring down Content parsing of other URLs. For reasons like these we achieve the functionality described in the previous section with the following microservices.

| Microservice | Characteristics | Category |
|---|---|---|
| URL Frontier | 1. Compute Intensive<br>2. On Disk Column oriented Storage for URL priority table | Stream Processing |

| | | |
|---|---|---|
| | 3. Key-Value store for URL Host to queue/worker mapping | |
| HTML Downloader | 1. Network Intensive<br>2. Stateless workers<br>3. Object store to store downloaded content and Crawl state<br>4. In-memory cache for popular URLs | Key-value |
| Content Parser | 1. Compute Intensive In memory Parser | Stream Processing |
| ContentSeen | 1. Compute Intensive, In memory Hash Calculation and Deduplication of Content<br>2. Key Value Object store to store parsed Web content and hash(content) | Stream Processing |
| LinkExtractorAndFilter | 1. Compute Intensive, InMemory Parser<br>2. Compute Intensive URL filter rules | Stream Processing |
| URL Seen | 1. Compute Intensive in memory hash calculation<br>2. Key Value Object store to store URLs | Stream Processing |

## URL Frontier Microservice

This microservice covers the URL frontier. It uses the seed URLs passed to fetch the web page. Two strategies are possible to fetch the web page, either Depth First Search (DFS) or Breadth First Search (BFS).

Commonly, the web crawler uses the BFS graph algorithm to fetch the web pages. DFS is not adopted as it can go very deep with billions of web pages. However, the issue with BFS for crawling is that most links from a webpage are linked back to the same host ; for instance, URL's from *wikipedia.com* refer back to *wikipedia.com*. Thus BFS on *wikipedia.com* will bombard the *wikipedia.com* web server, which contradicts the **Politeness** factor. And if these requests are parallelized, then the web-server will be overloaded further to service such requests . This makes the approach more impolite and may cause the web server to deny crawl requests (Distributed Denial of Service).

Also a random URL parsed from an Apple/web-site discussion forum has a much lower priority in parsing than a URL parsed out from Apple/web-site home page. So, the URL's URLs coming out from the same host need to be assigned different **Priorities** (when to crawl).

The general idea of politeness is to download one page at a time from the same host. A delay can be added between two download tasks going to the same URL host. However, if the URL hostnames are mapped to different workers, implementing a delay on crawl requests to a specific URL hostname is difficult since tracking downloads to the same URL host becomes a distributed problem. In order to avoid this distributed tracking problem, the URL host names

need to be mapped to specific download workers so that the download worker always puts a delay between two subsequent crawl() requests.
Traditional BFS do not support priorities. Putting the discovered web-pages during the graph traversal into a priority-queue and then picking them from the queue makes it into a form of Dijkstra or Prim's or A* or Best-first kind of graph traversal solution for the Leetcode 1236 code above.

URL frontier employ queues to implement the **Priority** and **Politeness** of crawling

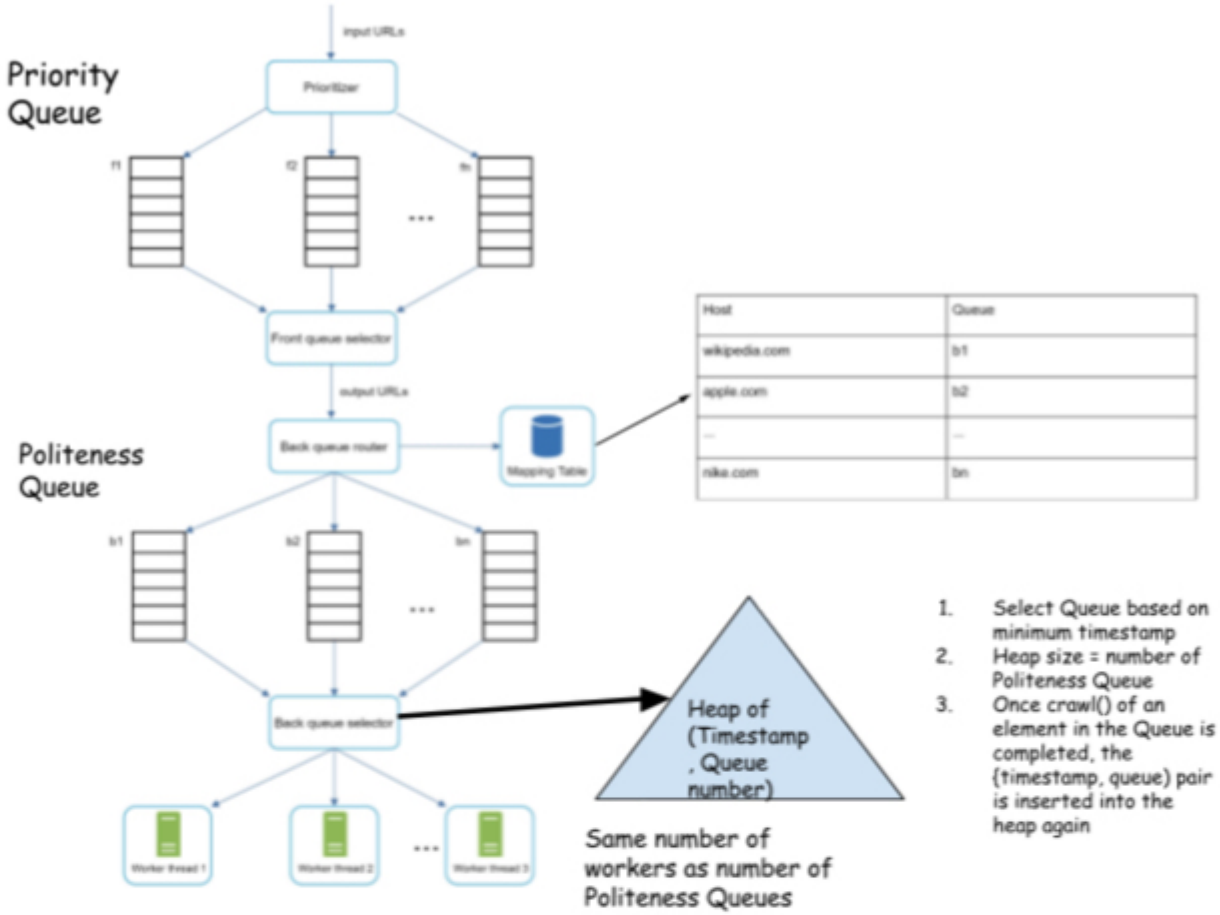
1. **Priority** assignment : we call this the **Priority Queue** which manages priorities.
    a. The Prioritizer assigns priority for URLs from the URL frontier
    b. Queues f1 to fn has an assigned priority
    c. Queue selector randomly chooses a queue f1..fn with bias towards queues with higher probability.
2. **Politeness**: Introduce delays between crawl() requests to the same hostname using a **Politeness Queue**. For this,
    a. Total number of queues b1, b2, ..bn is exactly the same as the number of stateless workers (HTML Downloaders)
    b. Each URL host is assigned a specific queue. This is the Mapping Table in the picture below and is depicted in the Back Queue part of the figure below to manage politeness.
    c. Once all queue elements of a specific URL are processed, the queue may be assigned to a different URL host

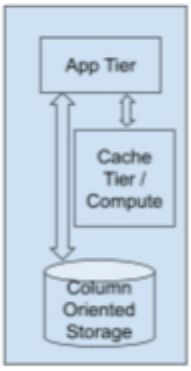
Priority or weightage to the URLs are assigned based on:

1. The web traffic that hits this URL
2. How frequently this page gets updated
3. How frequently the pages is accessed
4. Overall Page Rank
5. Update Status of the Page

The possible **Url-Priority Table** design to accommodate the above needs would be:

| **URL** | **Frequency of Access** | **% of Web Traffic** | **Last updated** | **Politeness Quotient** | **Status of Page** | **Page Rank** |
|---|---|---|---|---|---|---|
| Up to 2048 characters | | | | | | |




The App Tier Logic can be further broken down into:

1. Threads to
    a. prioritize input URLs using the URL-Priority table
    b. and place URLs in URL frontier queues
2. The DB Tier provides
    a. persistent column oriented storage access to the URL-Priority table entries since some columns are frequently updated and read.
    b. access to a key-value mapping table of Politeness Queue number to URL host.

The characteristics of this microservice

1. The number of URLs in the URL frontier could be hundreds of millions.
2. The number of URLs in the queues could be in the hundreds of millions.
3. There are multiple consumers of the web content and the producers and the

consumers operate at different rates

4. The rate of consumption of the queue entries are different from the rate at which they are produced.
5. For this streaming system we employ specific topics in a kafka queue infrastructure.

Given the above characteristics, we employ

1. Kafka queues to implement the URL frontier
2. Kafka queues to implement the front queue (prioritization) and back queue (storing URLs for workers)
3. And one of the consumers of the web content is the next microservice in sequence HTML Downloader. This microsevice could be a Kafka listener.

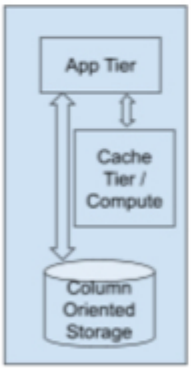
## HTML Downloader Microservice

With DNS resolution included, this API could be a bottleneck since most DNS lookup libraries do not allow simultaneous dns lookups. Also the latency of DNS lookup is high. To mitigate the issue

1. we cache the DNS results.
2. we create an asynchronous API underneath with the politeness factor included.

With a commodity server having 8 threads per core, 1000/x millisecond per task, and at 30% load factor, the throughput would be 30000/x.

In the App Tier with DNS access, x could be considered to be around 5ms P99, which leads to 6000 qps. With DNS caching provided for an already seen URL, the DNS query time could be considered to be minimized to 2ms. With 2ms value of x, it results in 15000 qps.




The App Tier Logic can be further broken down into:

1. Worker threads pick up URLs form the URL frontier
    a. The crawl jobs are distributed across multiple servers
    b. The crawl jobs are spawned on Geo-locality of the URLs hosts to get a faster download experience
    c. Consistent Hashing is employed to distribute load among workers
2. The Cache tier caches URL content thus providing fast access for the next Content Parser microservice for popular URLs
3. The storage tier is
    a. persistent key value storage mapping URL to web content
    b. stores the status of the download

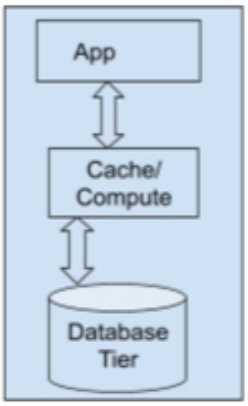
Once the web content is downloaded successfully a notification is sent on the Kafka topic with a pointer in the DB where the web content resides. ContentParser microservice picks up the processing forward.

### API Design

1. crawl(seedUrl, status)

## ContentParser Microservice

This microservice verifies and validates the HTML content that has been downloaded in the previous step. The various validations in the App tier include:




1. Normalize HTML content
2. Reject Malicious content
3. Reject Objectionable content
4. Reject Proprietary content , code snippets
5. Reject Advertising content : data noise
6. Removes Spider Trap : URL causing infinite crawl ; avoided by enforcing a maximal length of URL (example : www.aaa.com/a/b/a/b/a/b/a/b/.....)

This again is a compute intensive microservice that can be parallelized at the API level for throughput and efficiency. It uses the same cache and DB tier from the HTML Downloader microservice. Once the parsing and rendering of the web content is completed it posts a notification in a queue (Kafka topic) for the next microservice in the sequence ContentSeen to pick up the processing.

### API Design

1. parseUrl(url, status)

## ContentSeen Microservice

This microservice verifies if the HTML content is already present in the storage and optimizes duplicate parsing and resources consumed thus.

1. If it determines that the content is already in storage, then no further processing is done on this page and eliminates data and processing redundancy.
2. The way the HTML content is validated is calculating hash values of the page. These hash values are compared for presence or absence of the already visited page content.
3. When new content is to be validated again, it first checks if it is present in the cache, and then queries the database.

4. If the page content is absent then this data is passed to the LinkExtractorAndFilter Microservice

The storage layer for the Content Storage could be an object store. The Storage layer for the Content Seen microservice could be a key-value store {Hash, <set of urls>}.

Once the ContentsSeen is done, a notification is put by this microservice into a queue (kafka topic) for the next microservice in sequence LinkExtractor to pick up the processing.

## Bloom Filter

Since there are millions of web-pages at a time, storing these hash-values themselves is not possible in memory. In such cases a bloom filter can be used. Note that a bloom filter will save space but it is a probabilistic data structure : it may say a URL is present while it is not. Saving the hash of all URLs and comparing is a full proof but expensive strategy. But with enough hash functions and bits, the bloom filter can be made more accurate.

Some description of Bloom filter is below :

Data structure to efficiently check whether an element belongs to a set

Imagine a set of integers. To check whether an element is present in the set

- Brute force:
  - Time complexity: O(n) scan, O(nlogn) sort and O(logn) binary search
- Full blown bitmap
  - Time complexity: O(1)
  - Space complexity: Bitmap of O(INT_MAX) space
  - Given an element, check bitmap
  - Not feasible

- Use a set of k Hash functions
- Maintain a bitmap of size N
- Each hash function maps an element to a value between 0 and N-1
- Preprocessing: createFilter(Set)
  - For each integer in the set,
    - For each hash function
      - Apply the function and get a value X, 0<= X <= N-1
      - Bit set position X in the bitmap
- Runtime: checkPresence(element)
  - For each hash function
    - Apply the function and get a value X, , 0<= X <= N-1
    - Check if X is set in the bitmap
  - If all the bits from the above are set, the element **may or may not be present** in the set
  - If any of the bits is not set, the element **is definitely not present** in the set

Since the Bloom filter can tell definitely if an element is NOT present, it can be used to decide whether the content is NOT present.

### API Design

1. CheckContentInStorage(input_page, status)
2. CalculateHash(input_page)
3. ComparePageHash(input_page, stored_page)

To demonstrate the concept, an implementation of a simple bloom filter is done below in O(n) time and O(1) memory to detect a duplicate number in a given array. The real benefit is seen when there is a large number of streaming int or strings.

The question is around leetcode 287 :

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only one repeated number in `nums`, return *this repeated number*.

```
class Solution {
public:

    // large number of bits to reduce collison and increase accuracy
    #define BITS_SIZE (1024 * 1024 * 5)

    std::bitset<BITS_SIZE> bit_masks = {0};

    int64_t hash_fn1(string s)
    {
        int64_t hash = 1;
        for (int i = 0; i < s.size(); i++)
        {
            hash = hash + pow(19, i) * s[i];
            hash = hash % BITS_SIZE;
        }

        return hash % BITS_SIZE;
    }

    int64_t hash_fn2(string s)
    {
        int64_t hash = 7;
        for (int i = 0; i < s.size(); i++)
        {
            hash = (hash * 31 ^ s[i]) % BITS_SIZE;
```

```
        }
        return hash % BITS_SIZE;
    }

    int hash_fn3(string s)
    {
        int64_t hash = 3;
        int p = 7;
        for (int i = 0; i < s.size(); i++) {
            hash += hash * 7 + s[0] * pow(p, i);
            hash = hash % BITS_SIZE;
        }
        return hash % BITS_SIZE;
    }

    int hash_fn4(string s) {
        vector<int> primes{ 3, 7, 11, 13, 17};
        int64_t hash = 0;

        for (int hash_id=0; hash_id < primes.size(); ++hash_id) {
                for (int i = 0; i < s.size(); i++) {
                    hash = (primes[hash_id]*hash ^ s[i]) % BITS_SIZE;
                }
        }

        return hash % BITS_SIZE;
    }

    int findDuplicate(vector<int>& nums) {

        for (int i=0; i < nums.size(); ++i) {

            std::string s = std::to_string(nums[i]);

            // int a = h1(s);
            int a = hash_fn1(s);
            int b = hash_fn2(s);
            int c = hash_fn3(s);
            int d = hash_fn4(s);

            if ( bit_masks[a] && bit_masks[b] && bit_masks[c] && bit_masks[d]) {
                // most likely present
                return nums[i];
            }

            bit_masks.set(a);
            bit_masks.set(b);
            bit_masks.set(c);
```

```
            bit_masks.set(d);
        }

        return (-1);
    }
};
```

## LinkExtractorAndFilter Microservice

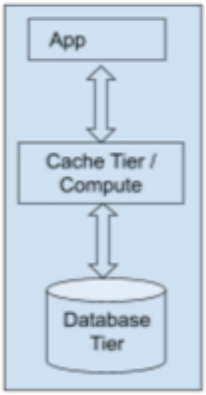
This microservice covers :


1. The Links are extracted from the HTML page
2. It is validated against the Robots.txt rules.
3. The content is parsed through the Filter rules checks and then the Urls are put into a queue (kafka topic) to be picked up by the next microservice UrlSeen .

### API Design

1. extractURL(htmlPage)
2. validateURL(url, status)
3. filterURL(url, status)

## UrlSeen Microservice




The URL seen checks if the URL is already in the URL frontier. It uses a hash of the URL and then a bloom filter constructed from the all existing URL frontier hash values to check if the URL does not exist in the URL frontier. If it does not exist it adds the URL to the URL frontier and updates the DB. The storage layer here is a key-value row-oriented storage.

### API Design

4. urlSeen(url, status)
5. updateDB(url, status)

## Design for Scale

For high performance,

1. If a URL host does not respond to crawl within a specified timeout, the crawl job can time out, update the status and move on. If it happens repeatedly, the URL can be put in an extremely low priority front queue.
2. To enable new content,
    a. plugins can be added to download PNG files as below
    b. Wen monitor can be added detect copyright infringements


The web crawler system is an AP system with eventual consistency. There are no strict CP requirements in the system.

## Future / Further Enhancements

There is lot of studies that have been undertaken in the enhancement to crawling include:

a. Incremental Crawling Support
b. User Behavior based Crawling
c. Focused crawlers
d. IoT Based Web Crawler