Introduction

Web crawler is used to gather the new content available on the internet. A web crawler starts by collecting the data from the web pages and then follows the link referenced by the web pages to collect new content.

Problem Requirements

- Scrape all the content available on the internet.
- Store the crawl content.
- Respect the crawling policy of the website. This is present in robot.txt file of the website which has the information that what you can do and what you can't if you are a bot which is crawling my website.
- Complete this process in x week.

Capacity Estimation

- One billion pages to be crawled.
- Each page is of size 1MB.
- Total storage required is 1billion*1MB = 1PB.
- If x week = 1 week. Number of second is 7*24*60*60 ~ 600k which is around 1600 per second.
- ~~Considering over a decent network, each webpage takes 1 second to load , we will need 1600 threads to crawl. If each machine is of 4 cores, total machine required is 1600/4 = 400.~~

Process Overview

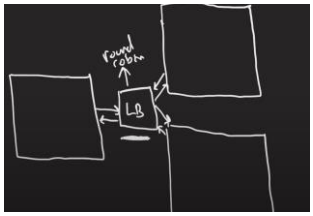
- Fetch the URL's from 'to-crawl' list.
- Check if the same URL has already been processed.
- Get the IP from the DNS of the host.
- Make an HTTP request to the IP.
- Check the host robots.txt for the crawling policy.
- Fetch the content.
- Check if the content has already been crawled and stored via a different URL.
- Parsed the content and store it.
- Added the reference web pages to the 'to-crawl' list.

Deep Dive

- **Fetch the URL's from 'to-crawl' list. Let's call it a frontier.**

    - **How to reduce the time it takes to fetch the URL from frontier.**
    - **We have to essentially reduce the network call. If the frontier is stored locally, it will be very fast.**
    - **There are some of the disadvantages associated with the above approach.**
      a. We may also end up processing duplicate URLs from 2 different nodes.
      b. Each local frontier can grow depending on the reference links we get from the web pages. Chances are nodes processing that frontier can end up doing lot of work as compared to other nodes which might be sitting idle.

    - **Solution of both the problems.**
      Distributed frontier.
      a. We want each node to process equal amounts of work. Instead of storing locally the next URL to be fetched, each node will send the URL to load balancer. Load balancer can then distribute the URL to nodes in round fashion.



b. Avoid duplicate fetching.
Easiest way is to store the fetched URL's information in a database.
Down merits are extra network call.
There is another smarter way to do this.
We are allotting the URLs via round robin fashion to the nodes. Optimized way can be -> If there are 2 URLs with the same host, they should be processed by same node. Essentially if hash(hostx) == hash(hosty), they should be processed by same node.
Each node is going to handle a hash range of host. Hash function should take care of uniform distribution.
Now each node can store the state locally to make the decision.

Avoid duplicate content on different sites.

Can we do a smart way to avoid extra network call to avoid ourself from processing duplicate content.

a. We have to fetch the content first.
b. We can take the hash/finger printing of the content.
c. Since there is no way to know which nodes has processed the content as the request are getting distributed using the hash of the URL, we have to use centralized way to see if we have seen this hash before.

How can we make this faster?

- **Content Hash Checking**
  - **Since duplicate content can be shown on any of the nodes, we need some centralized way to check if we have already this this hash.**
  - **If the hash function if 64 bits, and we have to at max hash 1 billion pages, it will take around 64bits*1000000000 = 8GB.**
  - **Store this as redis hashes. Lookup will be O(1) .**
    We can have a slave in the redis to make this system as fault tolerant. If we are reading and writing from the leader, we can also guarantee the strong consistency.
    If our leader goes down, we end up losing some write and as long as we are okk for processing 2 same content, we will okk in this case.

- **Domain Name System**

  Provides us a mapping of host to IP address. Requires a network call. How can we optimize this.
  - **In previous section, each node was processing the content from the same node.**
  - **We can cache the DNS request on each node. We can use LRU cache eviction to maintain the memory constraint.**

- **Robot.txt**
  - **We can also cache the crawling policy as each node is processing the request from same host.**
  - **E.g Don't crawl twitter.**
    In this case, node will ignore and fetches the next URL from frontier.
  - **You can only crawl twitter each minute**
    Each node can maintain the table having timestamp of last time you crawled the host.
    If it' less than 1 minute, we can push the crawl url back to frontier.

- **Fetching the URL's content.**
  Can we do something to speed this up. Can we have the server in UK if the URL have .uk in it.
  Unfortunately, not as it doesn't mean the server resides in UK for this.

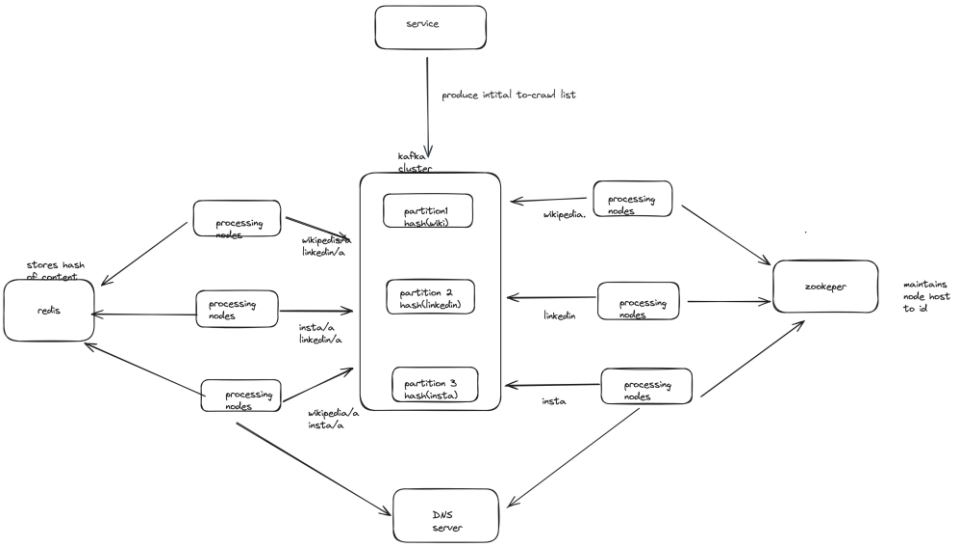
- **Storing Results**
    - **We want data locality. The data written to the server should be close to the nodes.**
    - **We can use S3 or azure blob in the same region (if possible same data centre) in which our nodes are running.**

Pushing to Frontier

- How should frontier store the data? Which data structure it should use.
  We can use Queue with FIFO. We can also use priority queue in case we need to add the priority of the URL's.

Archictecture choices.

- We want reliability. We don't want any website to skip.
- We can use Kaffka as frontier where we don't have to write our own partioning logic.



a. **Processing nodes on left- and right-hand side are same. Same node is working as consumer and producer.**

b. **Number of partitions = Number of consumers.**
c. **Consumer count is 1. It denotes the number of consumers getting attached from the same instance of service for a given consumer group.**
d. **Kafka partition id is host of the website to be crawled.**
e. **We need to make sure each instance of processing nodes is consuming from a single partition only. I.e. 1:1 mapping of the partition with the processing nodes.**
f. **In zookeeper we maintain the information of Kafka partition id and allotment of this partition id to particular node whenever the processing node registers itself with the zookeeper. Zookeeper will make sure to only allocate the partition id which is not allocated to the node.**
g. **Our Kafka partition assignment strategy will get the partition id we got from zookeeper and start consuming from that partition.**