Crawler

Requirements

Core

- Robutness: The Web contains servers that create spider traps, which are generators of web pages that mislead crawlers into getting stuck fetching an infinite number of pages in a particular domain. Crawlers must be designed to be resilient to such traps. Not all such traps are malicious; some are the inadvertent side-effect of faulty website development.
- Store HTML pages only? Or need other types of media such as images and videos
- What protocols we support: HTTP/HTTPS/FTP

Optional

- Politeness: Web servers have both implicit and explicit policies regulating the rate at which a crawler can visit them. These politeness policies must be respected.
- RobotsExclusion: The Robots Exclusion Protocol requires a Web crawler to fetch a special document called robot.txt which contains these declarations from a Web site before downloading any real content from it.

- Performance and efficiency: The crawl system should make efficient use of various system resources including processor, storage and network bandwidth.
- Quality: Given that a significant fraction of all web pages are of poor utility for serving user query needs, the crawler should be biased towards fetching “useful” pages first.
- Freshness: In many applications, the crawler should operate in continuous mode: it should obtain fresh copies of previously fetched pages. A search engine crawler, for instance, can thus ensure that the search engine’s index contains a fairly current representation of each indexed web page. For such continuous crawling, a crawler should be able to crawl a page with a frequency that approximates the rate of change of that page.
- Extensible: Crawlers should be designed to be extensible in many ways – to cope with new data formats, new fetch protocols, and so on. This demands that the crawler architecture be modular.
- Distributed: The crawler should have the ability to execute in a distributed fashion across multiple machines.
- Scalable: The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth.

Estimation

Crawl target

- Entire web. Suppose 1 trillion web pages
  - 10^12 / 7 / 86400 = 1.6M / s

Smaller goal

- 15B within four weeks

Specific goal

- Crawl 15 billion pages within four weeks

RPS estimation

How many pages do we need to fetch per second

```
15B / (4 weeks * 7 days * 86400 sec) ~= 6200 pages/sec
```

How many operations need to be performed

- For every link to a page encountered, the following operations are needed:
    i. Check if this page is already in the data store
    ii. Insert the link into the queue
- ops = dps * (2 * elpd) + is
    - ops = back-end i/o operations per second
    - dps = number of processed page per second. The initial target for the crawler are 6200 pages per second (about 864.000 domains per day)
    - elpd = external links per page. Information about this number can be found in several other papers. For our calculation, we assume an average of 7.5 external links per web page.
    - is = amount of operations needed for storing the collected information. depending on the back-end, this might only be 1 write operation (e.g. a single SQL INSERT)
- With these numbers, we end up with approximately
    - $6200 * (2 * 7{:}5) + 1 + 2 * (0{:}005) \approx 93, 620$ operations per second on our back-end system just for crawling the front page alone.

Storage estimation

- Page sizes vary a lot, but if we will be dealing with HTML text only, let's assume an average page size of 100KB.

```
15B * 100KB  ~= 1.5 petabytes
```

- Assuming a 70% capacity model, total storage we will need:

```
1.5 petabytes / 0.7 ~= 2.14 petabytes
```

Design

Single threaded crawler

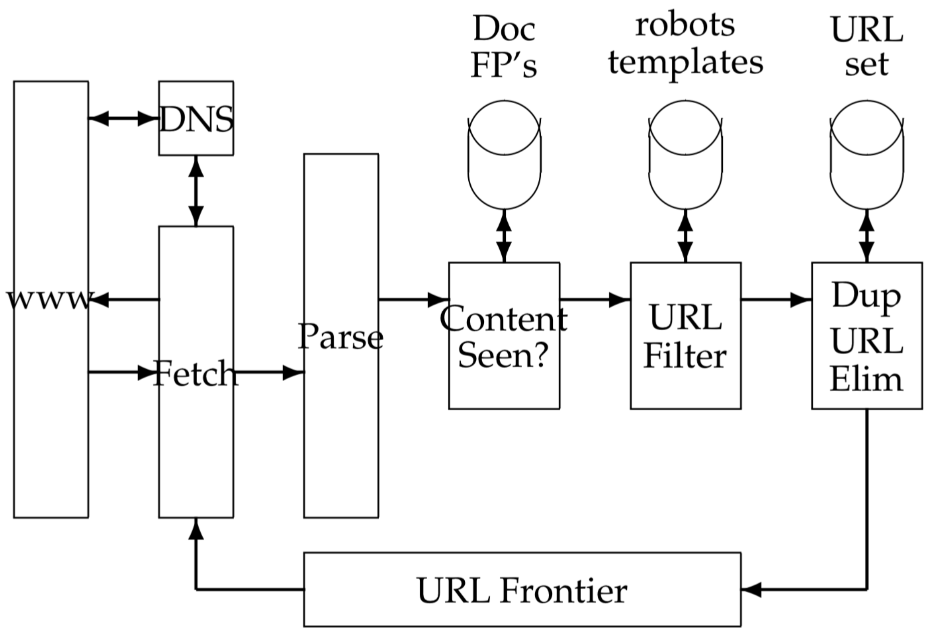
- Input: Url seeds
- Output: List of urls
- Producer-consumer implementation in Python

```
// breath first search, single-threaded crawler
function run
    while ( url_queue not empty )
        url = url_queue.dequeue()
        html = web_page_loader.load( url ) // consume
        url_list = url_extractor.extract( html ) // produce
        url_queue.enqueue_all( url_list )
    end
```

A multi-threaded web crawler

- How different threads work together. See appendix for detailed programs.
  - sleep: Stop a random interval and come back to see whether the resource is available to use.
  - condition variable: As soon as the resource is released by other threads, you could get it immediately.
  - semaphore: Allowing multiple number of threads to occupy a resource simultaneously. Number of semaphore set to 1.
- However, more threads doesn't necessarily mean more performance. The number of threads on a single machine is limited because:
  - Context switch cost (CPU number limitation)
  - Thread number limitation
    - TCP/IP limitation on number of threads
  - Network bottleneck for single machine

Url filter and prioritization



-

1. Starts with taking a URL from the frontier and fetching the web page at that web page.
2. The page is parsed and the link within it is extracted.
3. Each extracted link goes through a series of tests to determine whether the link should be added to the URL frontier.
    - First, the thread tests whether a web page with the same content has already been seen at another URL.
        - The simplest implementation for this would use a simple fingerprint such as a checksum.
        - A more sophisticated test would use shingles instead of fingerprints. (What is Shingles ???)
        - Bloom filter. A Bloom filter is a probabilistic data structure and is used for answering set-existential questions (eg: has this URL been crawled before?). Due its probabilistic nature, it can give erroneous results in the form of false positives. You can however tweak the error rate, allowing for only a small number of false positives. The great benefit is the large amount of memory you can save (much more memory efficient than Redis Hashes). If we start crawling pages in the hundreds of millions, we definitely would have to switch to this data structure. As for the false positives, well, there ain't no harm in occasionally crawling the same page twice.

    - Next, a URL filter is used to determine whether the extracted URL should be excluded from the frontier based on one of several tests. For instance, the crawl may seek to exclude certain domains (say, all .com URLs) – in this case the test would simply filter out the URL if it were from the .com domain.
        - Many hosts on the Web place certain portions of their websites off-limits to crawling, under a standard known as the Robots Exclusion Protocol. This is done by placing a file with the name robots.txt at the root of the URL hierarchy at the site. Here is an example robots.txt file that specifies that no robot should visit any URL whose position in the file hierarchy starts with /yoursite/temp/, except for the robot called "searchengine".
    - Then a URL should be normalized. Often the HTML encoding of a link from a web page p indicates the target of that link relative to the page p.

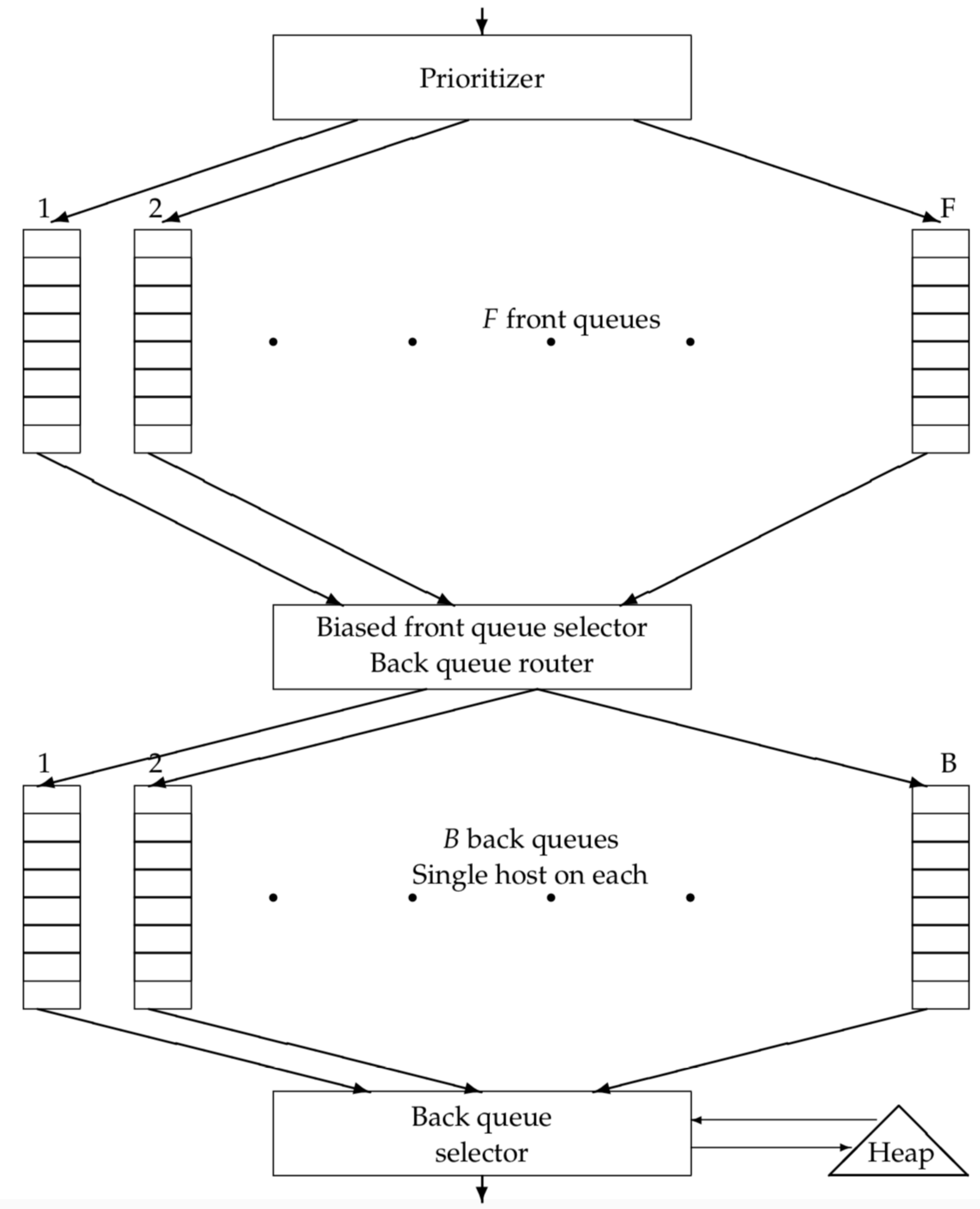
Scale the queue

Distributed web crawler

- URL queue is inside memory. Queue is too big to completely fit into memory. Use a MySQL DB task table
    - state (working/idle): Whether it is being crawling.
    - priority (1/0):
    - available time: frequency. When to fetch the next time.

id	url	state	priority	available_time
1	"http://www.sina.com/"	"idle"	1	"2016-03-04 11:00 am"
2	"http://www.sina1.com/"	"working"	1	"2016-03-04 12:00 am"
3	"http://www.sina2.com/"	"idle"	0	"2016-03-14 02:00 pm"
4	"http://www.sina3.com/"	"idle"	2	"2016-03-12 04:25 am"

URL frontier



- A set of front queues: Prioritization
    - A prioritizer first assigns to the URL an integer priority i between 1 and F based on its fetch history (taking into account the rate at which the web page at this URL has changed between previous crawls).
        - Frequency of change: For instance, a document that has exhibited frequent change would be assigned a higher priority.
        - Other heuristics (application-dependent and explicit) – for instance, URLs from news services may always be assigned the

highest priority.

- Now that it has been assigned priority i, the URL is now appended to the ith of the front queues
- Two important considerations govern the order in which URLs are returned by the frontier.
    - First, high-quality pages that change frequently should be prioritized for frequent crawling. Thus, the priority of a page should be a function of both its change rate and its quality (using some reasonable quality estimate). The combination is necessary because a large number of spam pages change completely on every fetch.
    - We must avoid repeated fetch requests to a host within a short time span. The likelihood of this is exacerbated because of a form of locality of reference: many URLs link to other URLs at the same host. A common heuristic is to insert a gap between successive fetch requests to a host that is an order of magnitude larger than the time taken for the most recent fetch from that host.
- An importance score will be assigned to each URL which we discover and then crawl them accordingly. We use Redis sorted sets to store the priority associated with each URL and hashes to store the visited status of the discovered URLs. This, of course, comes with a large memory footprint.

- A set of back queues: Politeness

    - Each of the B back queues maintains the following invariants:
        - (i) it is non- empty while the crawl is in progress
        - (ii) it only contains URLs from a single host
    - An auxiliary table T is used to maintain the mapping from hosts to back queues. Whenever a back-queue is empty and is being re-filled from a front-queue, table T must be updated accordingly.
    - Process
        a. A crawler thread requesting a URL from the frontier extracts the root of this heap and (if necessary) waits until the corresponding time entry te.
        b. It then takes the URL u at the head of the back queue j corresponding to the extracted heap root, and proceeds to fetch

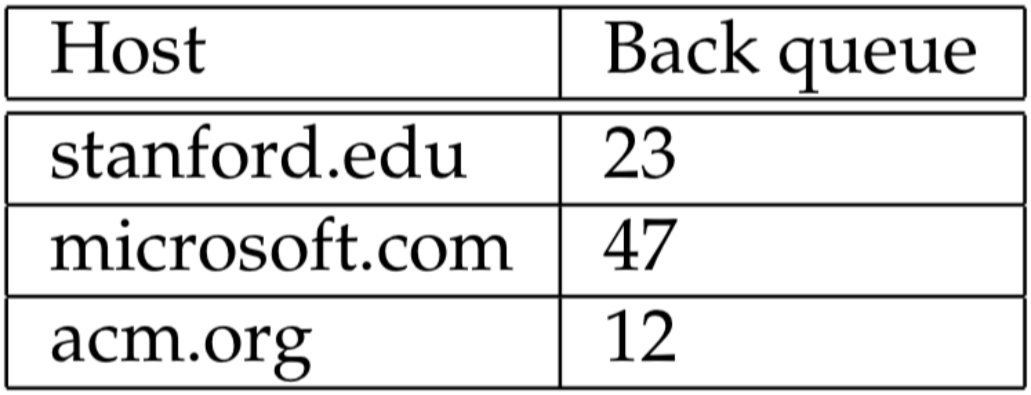
the URL u.

c. After fetching u, the calling thread checks whether j is empty.

d. If so, it picks a front queue and extracts from its head a URL v. The choice of front queue is biased (usually by a random process) towards queues of higher priority, ensuring that URLs of high priority flow more quickly into the back queues. We examine v to check whether there is already a back queue holding URLs from its host.

e. If so, v is added to that queue and we reach back to the front queues to find another candidate URL for insertion into the now-empty queue j.

f. This process continues until j is non-empty again. In any case, the thread inserts a heap entry for j with a new earliest time te based on the properties of the URL in j that was last fetched (such as when its host was last contacted as well as the time taken for the last fetch), then continues with its processing. For instance, the new entry te could be the current time plus ten times the last fetch time.



Host	Back queue
stanford.edu	23
microsoft.com	47
acm.org	12

DNS resolution

- DNS resolution is a well-known bottleneck in web crawling. Due to the distributed nature of the Domain Name Service, DNS resolution may entail multiple requests and round-trips across the internet, requiring seconds and sometimes even longer. Right away, this puts in jeopardy our goal of fetching several hundred documents a second.

    - A standard remedy is to introduce caching: URLs for which we have recently performed DNS lookups are likely to be found in the DNS cache, avoiding the need to go to the DNS servers on the internet. However, obeying politeness constraints limits the of cache hit rate.
    - https://nlp.stanford.edu/IR-book/pdf/20crawl.pdf for more details.

Scale by functional partitioning

- Crawler service
- Task service
- Storage service

How to handle update for failure

- Exponential back-off
    - Success: crawl after 1 week
    - no.1 failure: crawl after 2 weeks
    - no.2 failure: crawl after 4 weeks
    - no.3 failure: crawl after 8 weeks

How to handle dead cycle

- Too many web pages in sina.com, the crawler keeps crawling sina.com and don't crawl other websites
- Use quota (10%)

Multi-region

- When Google's webpage crawls China's webpages, it will be really really slow. Deploy crawler servers in multiple regions.

Appendix - Threading programs

Initial implementation

- Problem: At some point, consumer has consumed everything and producer is still sleeping. Consumer tries to consume more but since queue is empty, an IndexError is raised.

- Correct bnehavior: When there was nothing in the queue, consumer should have stopped running and waited instead of trying to consume from the queue. And once producer adds something to the queue, there should be a way for it to notify the consumer telling it has added something to queue. So, consumer can again consume from the queue. And thus IndexError will never be raised.

```
from threading import Thread, Lock
import time
import random

queue = []
lock = Lock()

# Producer keeps on adding to the queue
class ProducerThread(Thread):
    def run(self):
        nums = range(5) #Will create the list [0, 1, 2, 3, 4]
        global queue
        while True:
            num = random.choice(nums) #Selects a random number fro

            # queue is kept inside lock to avoid race condition
            lock.acquire()
            queue.append(num)
            print "Produced", num
            lock.release()

            time.sleep(random.random())

# Consumer keeps on removing from the queue
class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:

            # queue is kept inside lock to avoid race condition
            lock.acquire()
            if not queue:
                print "Nothing in queue, but consumer will try to
            num = queue.pop(0)
            print "Consumed", num
            lock.release()

            time.sleep(random.random())

# start one producer thread and one consumer thread
ProducerThread().start()
ConsumerThread().start()
```

Improve with Condition

- Condition object allows one or more threads to wait until notified by another thread. And this is exactly what we want. We want consumer to wait when the queue is empty and resume only when it gets notified by the producer. Producer should notify only after it adds something to the queue. So after notification from producer, we can be sure that queue is not empty and hence no error can crop if consumer consumes.
    - Condition is always associated with a lock
    - A condition has acquire() and release() methods that call the corresponding methods of the associated lock. Condition provides acquire() and release() which calls lock's acquire() and release() internally, and so we can replace lock instances with condition instances and our lock behaviour will keep working properly.
    - Consumer needs to wait using a condition instance and producer needs to notify the consumer using the condition instance too. So, they must use the same condition instance for the wait and notify functionality to work properly.

```
from threading import Condition

condition = Condition()

class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            condition.acquire()

            # Check if the queue is empty before consuming. If yes
            # wait() blocks the consumer and also releases the loc
            # Now unless consumer is notified, it will not run.
            if not queue:
                print "Nothing in queue, consumer is waiting"
                condition.wait()
                print "Producer added something to queue and notif
            num = queue.pop(0)
            print "Consumed", num
            condition.release()
            time.sleep(random.random())

class ProducerThread(Thread):
    def run(self):
        nums = range(5)
        global queue
        while True:
            # Producer can acquire the lock because lock was relea
            condition.acquire()

            # Producer puts data in queue and calls notify() on th
            num = random.choice(nums)
            queue.append(num)
            print "Produced", num

            # Once notify() call is made on condition, consumer wa
            condition.notify()

            # Producer explicitly releases the lock by using condi
            condition.release()

            # And consumer starts running again. Now it will find 
            time.sleep(random.random())
```

Add a max size on the queue

```
from threading import Thread, Condition
import time
import random

queue = []
MAX_NUM = 10
condition = Condition()

class ProducerThread(Thread):
    def run(self):
        nums = range(5)
        global queue
        while True:
            condition.acquire()

            # Before putting data in queue, producer should check 
            if len(queue) == MAX_NUM:
                # If the queue is full, producer must wait. So cal
                # This gives a chance to consumer to run. Consumer
                print "Queue full, producer is waiting"

                # And then consumer should notify the producer.
                condition.wait()
                print "Space in queue, Consumer notified the produ

            # Once consumer releases the lock, producer can acquir
            num = random.choice(nums)
            queue.append(num)
            print "Produced", num
            condition.notify()
            condition.release()
            time.sleep(random.random())


class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            condition.acquire()
            if not queue:
                print "Nothing in queue, consumer is waiting"
                condition.wait()
                print "Producer added something to queue and notif
            num = queue.pop(0)
            print "Consumed", num
            condition.notify()
            condition.release()
            time.sleep(random.random())


ProducerThread().start()
ConsumerThread().start()
```

Use a queue instead

- Queue encapsulates the behaviour of Condition, wait(), notify(), acquire() etc.

```
from threading import Thread
import time
import random
from Queue import Queue

queue = Queue(10)

class ProducerThread(Thread):
    def run(self):
        nums = range(5)
        global queue
        while True:
            num = random.choice(nums)
            # Producer uses put available on queue to insert data
            # put() has the logic to acquire the lock before inser
            # Also put() checks whether the queue is full. If yes,
            queue.put(num)
            print "Produced", num
            time.sleep(random.random())


class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            # Consumer uses get.
            # get() acquires the lock before removing data from qu
            # get() checks if the queue is empty. If yes, it puts
            # get() and put() has proper logic for notify() too. W
            num = queue.get()
            queue.task_done()
            print "Consumed", num
            time.sleep(random.random())


ProducerThread().start()
ConsumerThread().start()
```

Reference

- blog post