# Whirlpool: Content Aggregation using N-node Distributed Web Crawler

*Report Compiled by:* Rihan Stephen Pereira
*Advisor:* Dr. Michael Soltys
MSCS Graduate 2018-2019

June 14, 2019

# Contents

# 1 | Introduction

A *web crawler* is a long running program that collects pages from the web and stores them to the disk either in raw format or after extracting its contents. Sometimes they are also referred as *random walker*, *worm*, *spiders*, *robot*. Crawlers serve variety of purposes. Web search engines are very well-known systems powered internally through crawlers. Other uses are web data mining for performing analytics, archiving portions of the web - a relevant active project is the Wayback Machine from The Internet Archive[19] operating since 2000. Most recently, there is an ongoing attempt from Software Heritage[20] to preserve publicly available source code.

This thesis is an attempt to implement a small-scale, special purpose crawler for collecting data on job postings from numerous job portals active on the web. This will help job seekers monitor openings of interest and act accordingly. This thesis does not deal with data mining but instead focuses on the design, implementation, and deployment of web crawler to perform data collection on given seed set. The crawler architecture builds on a open blueprint design of *Mercator*[8] which describes the design in enough detail but leaves out implementation specifics to developers.

- **Use of Message Queue:** In one case, the paper doesn't provides information on how the subsystems that make up a crawler are connected together. This thesis makes use of message broker to facilitate collaboration between components thus providing asynchronous, event-driven style of communication. It also highlights its strengths & weakness.

- **Crawl Ordering Problem:** A more crucial issue is without having a target to crawl, the problem of crawling becomes endless, infinite crawling even for specialized crawler such as this. This thesis discusses the crawl ordering strategy to shape its crawling only to specific content.

- **Distributing the crawl:** Furthermore, this thesis shows the shortcomings of linear hashing used for partitioning hosts to distribute the crawl and instead employs consistent hashing[13] algorithm and discuss its merits over it.

- **Development strategy:** Given this complex program at hand, it becomes obvious to separate the concerns by developing and testing each component in isolated pieces. This thesis leverages docker containers to build the project in stages. It allows to split the development and production environment by building images for the target platform to run the component.

- **AWS Services:** For running the crawler activity, this thesis makes the most of AWS 12-month free tier access by combining together suite of services and sketches an infrastructure to setup, deploy and demonstrate the program.
- **Microservices:** Lastly, this thesis explains the philosophy, principles behind microservices [15] by relating it to work done in this thesis.

## 1.1 Motivation

Implementing a web crawler can be seen as a fancy project and its complexity can easily be overlooked. Given the apparent simplicity of basic crawling algorithm. It posses numerous challenges. First of all, crawling the whole web is simply unrealistic even for the world's most advanced crawlers. Considering the current scale of web and while its still evolving, it is important to think thoroughly about characteristics of a crawler, because this brings control in the hands of developers, it enriches the value of problem they are trying to solve, it gives them a choice to either seek broad coverage over fresh content or maintain a balance between the two. Even a small scale version of a crawler can be built to be sophisticated and scalable. Moreover, building a crawler sets a playground to address problems such as concurrency, throughput, and load balancing. It is a program quite different from traditional, client-server paradigm that can fanout from single source file to modular, independent services that interact with each other through endpoints.

When discussing a system like crawler, people often critic saying that

"Stop worrying about scale. You're not Google/Amazon. Just use a relational database."

Well, that is true; building for scale without enough evidence from the metrics such as logging, locks you into an inflexible design. It is also a form of premature optimization. Along the same lines, there is also an urge felt to overengineer and loose focus on simplicity. Engineers loves to solve puzzles and challenge of building complex software. Overengineering is basically building a solution that is much more complex than is really necessary. This occurs when developer at a job tries to predict every possible use case and every edge case, eventually losing focus on the most common use cases.

However, the author of this thesis believes that it important to survey different technologies as each have their own strengths and weaknesses and choose the right tool for the job. The inclusion of message broker, docker, and AWS services to build the crawler are introduced with a mindset of promoting good design which will allow one to add more details and features later on. The design begins with a reasonable level of abstraction and with multiple iterations will gives better results.

# 2 | Background

This chapter primarily attempts to familiarize the reader with enough groundwork related to web crawler. It also elaborates on the miscellaneous resources that are integrated to support the objective of this thesis. It begins by explaining the basic steps taken by a crawler, followed by describing key features that underpin its system design. Properties of a crawler are discussed in the following section. A chronological walk through of highly established crawlers and key differences are noted later in the section. A section is dedicated to explaining the role of Message Queues. The remainder of this chapter covers different AWS services and docker used to interweave an environment for executing a complex program.

## 2.1 Basic Crawling Algorithm

The basic operation of any HTTP based crawler goes like this:

```
1: Let I ← {1,2,3,4,5} such that seed set S = {$U_i$| i $\in$ I}
2: $U_f \leftarrow$ S; where $U_f$ is a Frontier queue
3: procedure SPIDER($U_f$)
4:     while $U_f \neq \emptyset$ do
5:         $u \leftarrow$ Pop($U_f$)
6:         $p \leftarrow$ Fetch($u$)
7:         $T \leftarrow \exists p$ [{Extract($p, t$) | $t$ is a text }]
8:         $L \leftarrow \exists p$ [{Extract($p, l$) | $l$ is a link }]
9:         $U_f \leftarrow U_f \cup L$
10:        $\exists u$ [{Delete($U_f, u$) | $u$ is a already fetched URL}]
11: end
```

## 2.2 Features of a Crawler

The web is not a centrally hosted data warehouse but instead is comprised of billions of independent web hosts, operating within its limits. Taking this into consideration, a web crawler exhibits most of the following traits:

- **Being polite to web hosts:** A crawler shouldn't induce too much burden on target host at the time when they crawl. By not regulating this behavior, it is blocking the web site from pursuing its purpose or business and eventually there is a higher possibility of have ramifications on the operator running the crawler.

- **Resilent to spider traps a.k.a infinite loops:** Some hosts mislead crawlers into fetching page after page that never reach an end. Thus, a crawler should be robust to such fallacies.

- **Distributing load:** A crawler can make itself capable by distributing the load across multiple machines. Distributing the load across multiple machines is also a *shared nothing architecture*. It can make the crawler download web pages based on geographical proximity reducing request-response latency.

- **Scaling:** Scaling and Distributed may sound similar but subtle difference is a scalable crawler's sole purpose is to cop up with load as it comes, in this case, pacing the crawl rate as well as enhancing the crawl rate. With scalability in mind, the architecture needs to be rethought on every order of magnitude of load increase.

- **Page quality:** A crawler should seek broad coverage by being an explorer but at the same time it should be biased towards first selecting websites it is intended to crawl.

- **Extensible:** Like any complex program, a crawler system design should be flexible and extensible on top of its core modules.

## 2.3 Crawl Ordering Problem

The purpose of a web crawler can be viewed as traversing the web graph. The crawling order appears naturally to be breath-first problem (download pages following the links as they appear). But since the web is so massive and still expanding such a crawling activity is considered infinite and the crawler itself can be said to be *random walker* with no purpose. Most of the commercial or even open source crawlers have some sort of crawling order policy built into their system. It is a must for small-scale, special purpose crawler to order the extracted URLs because the act of acquiring content is restricted by various factors.

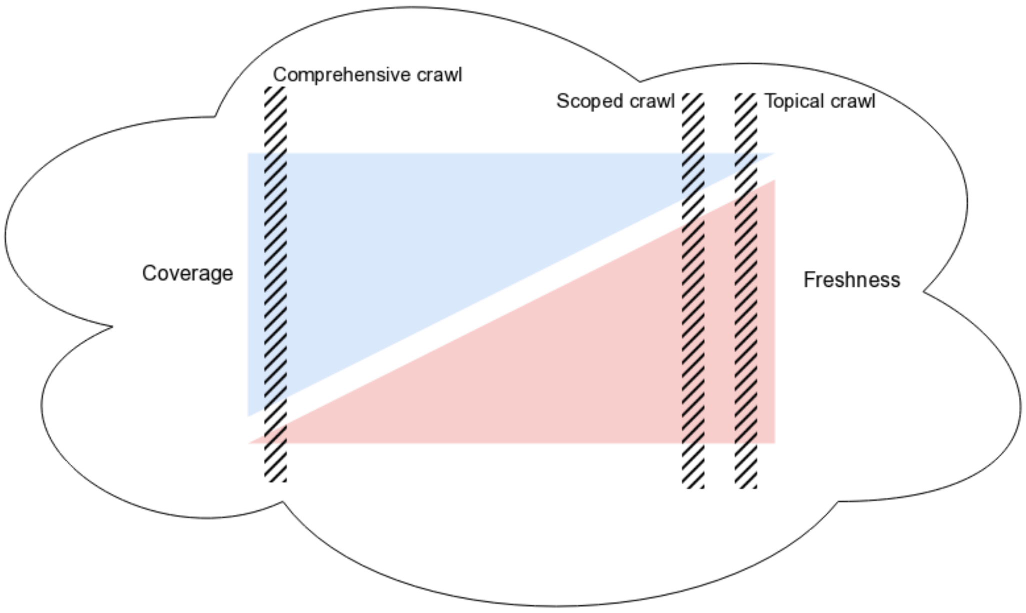


Figure 2.1: Crawl ordering based on Coverage vs. Freshness

**Coverage:** Focus is on fetching pages that crawler deems required.
**Freshness:** Here the focus is on revisting, redownloading already visited pages. Since web 2.0, pages are capable of dynamically changing content and popularity of Single Page Applications(SPA) has enabled rendering data through view logic controlled by javascript.

As shown in the figure 2.1, *Comprehensive Crawling* focuses on maximizing coverage by reaching out to content of all types. A particular page $p$ is weighted important by counting external links outside current site's perimeter. Crawl order policies like Breath-first Search, PageRank, and other variations are applied. Such activity is carried out by Commericial Search Engines.

On the other hand, *Scoped Crawling* narrows its crawling activity to certain portion of the web i.e only crawl a specific category (e.g everything about Stock Market). The crawling order is determined based on a degree to which a page $p$ falls under its umbrella. In constrast to Comprehensive, Scoped crawler is suitable for data-mining task by performing data collection on topic of interest.*Topical Crawling* is a form of Scoped Crawling in which page relevance is determined by - given a page $p$ links to another page $p'$ and is in-scope(within bounds) either directly or through a series of subsequent links. There are 3 techniques to order URLs to crawl.

1. Fish Search (Binary classificaiton). Here, the relevance is measured by treating links within visited page either relevant/ire-relevant upto a certain height.
2. The relevance is estimated by doing textual analysis on text surrounding a link in a visited page $p$ that points to yet-to-be visited page $q$.
3. Train and query a Machine Learning classifier to get the score of set of topics of interest to crawl.

This crawler has to balance its greediness level by considering page relevance. It is possible that a already relevant page $p$ may not always yield a new relevant page but it might would after going through several links of irrelevant pages. If the crawler is designed to be too greedy then it will have zero coverage which in turn means it won't discover new pages in its journey. A decay factor can be used to allow irrelevant fetches with an ultimate intention of finding relevant page of interest.

Finally, freshness is the attribute that makes the design of crawler being either continuous or non-continuous. The question of balancing content coverage and freshness in a crawler is solely a decision of entity behind it. A *batch crawling* design requires the restarted of a crawler at periodic interval to download revised content of previously crawled pages. A *continuous* or *incremental crawling* design would be technically a program that never terminates to revisit previously crawled pages.

## 2.4 Related Works

Web crawling is well studied topic, several crawlers emerged and vanished, some designs weren't open while some were not documented. This section briefly talks about history of web crawlers that have existed and have made notable design decisions each passing year.

**Wanderer** was the first crawler that was written in Perl in 1993. It ran on a single machine. It was used for collecting statistics and later used to power first search engine.

**MOMSpider** came into existence in 1994. It rate-limited requests to the web servers by introducing *Robot Exclusion Policy*. It allowed crawler operator to exclude sites. No focus on scalability. It used DBMS to store URL's and state of crawls.

**Internet Archive** [19] crawler addresses the challenges of growing web. It was designed to crawl 100 million URL's. Used disk-based queue to store URL's to-be crawled. Made use of bloom-filter to determine whether a page was previously crawled and if so, that page was not visited again. It also addressed politeness concerns and optimized *robots.txt* lookups. Apache Hendrix is a open-source out-of-box crawler used by the Internet Archive project entirely written in Java.

**Mercator** [8] came out in 1999. The paper discusses the blueprint design of its crawler. Initially it was non-distributed. It addressed social aspects of crawling through its frontier scheme. The announcement of 2nd paper incorporated host splitting component that made the crawler execution model distributed. It was extensively used in web mining projects. It ran on 4 machines for 17-days crawling over 891 billion pages. The program was designed with extensibility in mind and implemented fully in Java. The blueprint design is discussed in much detail in section 2.5 as it forms the basis of crawler designed for this thesis.

The **Polygot** web crawler also had distributed design. It had crawler manager handling multiple downloading process. It ran on 4 machines for 18 days and crawled over 120 million pages.

**IBM Webfountain** crawler was distributed, featured multi-threaded crawling process called 'ants'. It was polite and can be configured to change politeness policies on fly. Re-downloading of pages a.k.a freshness was based on historical change rate of pages. Webfountain was written in C++ & used Messaging Passing Interface(MPI) to allow collaboration between processes. It was deployed on 48 nodes.

**Ubicrawler** [9], 2004, was yet another scalable distributed web crawler. It employed consistent hashing algorithm to cope up with addition/removal of nodes designated for carrying out crawling task. The consistent hashing is explained in detailed in chapter 3 section 3.3. By using this technique it demonstrated graceful performance degradation in the event of failure. It was deployed on 5 nodes and dowloaded 10 million pages a day. This program was written entirely in Java.

**Multicrawler** [11], 2006, focused less on performance and crawl rate but more on detecting, transforming multiple formats of semantic web data.

**IRLbot** [10], 2009, is the most recent comprehensive crawler studied and implemented for over 3 years. It discusses its own implementation to address the increasing complexity of verifying unique URLs accumulated over time. Most notably it provides a solution to mitigate spider traps created by web sites. IRLbot was able to successfully crawled 6.3 valid HTML pages in a single day.

Apart from this, there exist open source web crawlers written in various languages notable ones are Apache Nutch(Java), Scrappy(Python).

## 2.5 Mercator Architecture

This section proceeds into detailed description of Mercator[8] web crawler surveyed in section 2.4. Figure 2.2 is a high-level composition of the same. As seen, Mercator specializes different steps defined in basic crawling algorithm covered in section 2.1 and adds several other steps to address the social and scalability challenges of web crawler. Its blueprint design offers horizontal scalability and therefore if implemented can run on more than one node. Each part that contributes to mercator crawler can also be referred to as a component, module, subsystem, etc. They can be used interchangeably from this point onward.

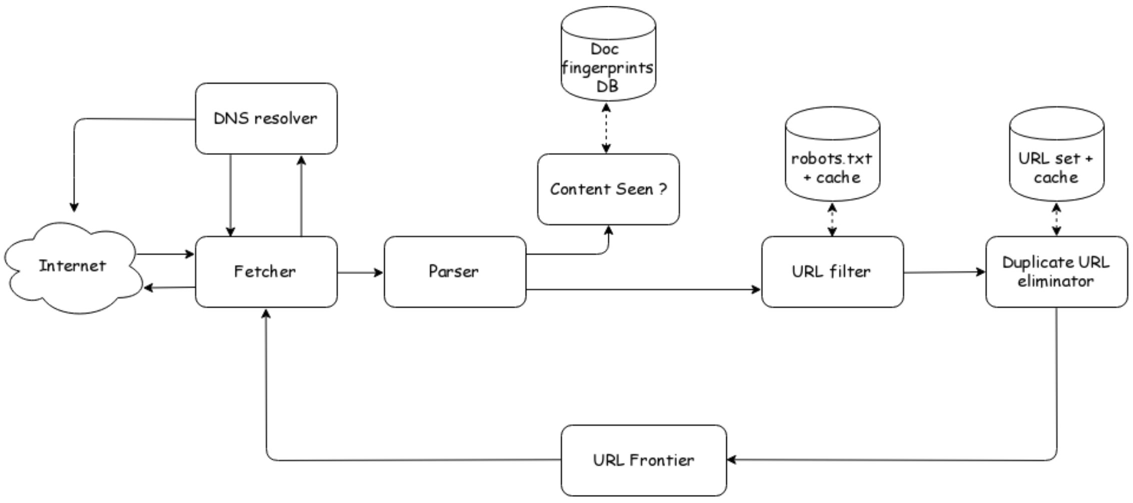


Figure 2.2: High-level organization of Mercator

Each work cycle begins and ends at the URL Frontier. Depending on what needs to be achieved, the entire crawler can run as a single process or can be partitioned into multiple processes - treating each subsystem as a process. Given a single URL, it goes through the cycle of being fetched followed by passing through various checks, filters and eliminations, then finally returned to the Frontier (for incremental crawling).

At the beginning of each logical loop cycle, a worker thread pops a URL from the Frontier data structure adhering to the priority and politeness policy. The output URL is then fetched by HTTP fetcher subsystem - which like any other web client contacts DNS module to get IP address of the corresponding web server. The web page is then retrieved and stored at a temporary location which is then picked up by parser subsystem which forwards the path to the page and extracted links to content seen and URL filter, respectively. The batch of URLs undergo a fixed set of pre-processing steps at the URL filter subsystem before being passed over to Duplicate URL eliminator (DUE). The DUE module tests each URL to determine whether the link should be added to the URL frontier.

### 2.5.1 Fetcher, Parser, and DNS

It occurred to designers of Mercator that the task of fetching each URL from seed set or newly discovered links is complex. First, the request made to the web server at a given URL endpoint will have several outcomes as a response. Therefore, it becomes a bottleneck to implemented fetcher as only single threaded, blocking synchronous module. Instead it should be written as a multi-threaded, synchronous I/O or non-block, asychronous I/O to speedup the operation. Moreover, it is also inefficient to have DNS resolve a given URL's IP address each time a fetch request is made to the same web server. This can be fixed by having a cache of lookup pair for each web host in the frontier. According to Mercator designers, another difficulty is the lookup implementation itself of DNS entries is blocking; meaning that at a time only one request is considered and completed and all other requests queue up. The solution for this is yet again a multi-threaded approach or a asynchronous I/O wrapper.

### 2.5.2 Handling De-duplication

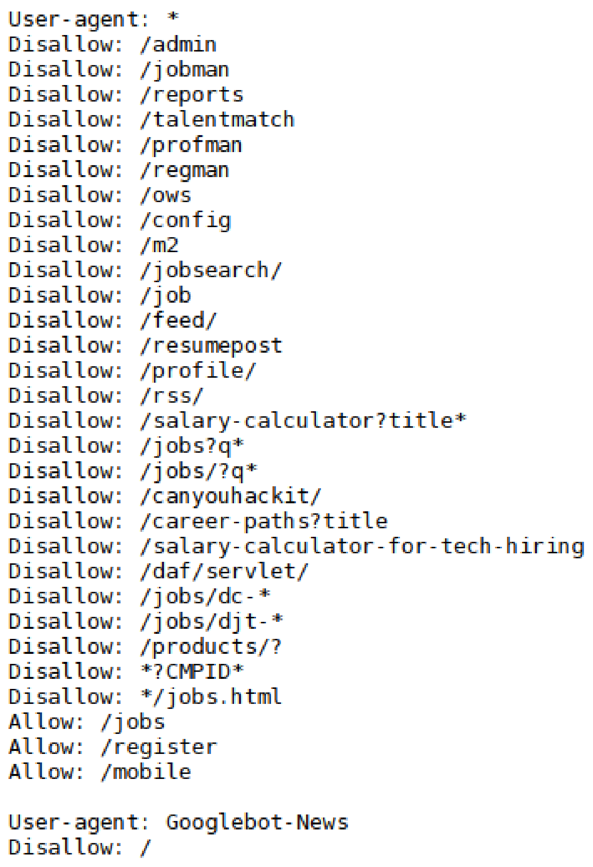
Any professional or sophisticated crawler has the capability to minimize De-duplication. De-duplication simply tests whether a web page with the same content has already been seen at another URL. As per figure 2.2, the content seen subsystem takes care of the same. There exist three established methods to solve this problem.

1. Document fingerprinting (checksums)
2. Bloom Filters
3. Shingles

(*this will be expanded further*)

### 2.5.3 URL Filtering

The URL Filter component takes input a batch of extracted URLs, applies a series of tests, modifiers on each URL, composes a newer batch of URL which fall in its criteria and forwards it to the next subsystem. For instance, assume a set of links $\sum_{n=1}^{10} U_n$ extracted from page $p$, there is a possibility that some $U$'s link are relative to the page $p$, in such cases, this component normalizes such $U$'s turning them into absolute URLs. Also, the crawler can enforce a rule to exclude out of bound URls - those that don't fall within a given list of domains.



```
User-agent: *
Disallow: /admin
Disallow: /jobman
Disallow: /reports
Disallow: /talentmatch
Disallow: /profman
Disallow: /regman
Disallow: /ows
Disallow: /config
Disallow: /m2
Disallow: /jobsearch/
Disallow: /job
Disallow: /feed/
Disallow: /resumepost
Disallow: /profile/
Disallow: /rss/
Disallow: /salary-calculator?title*
Disallow: /jobs?q*
Disallow: /jobs/?q*
Disallow: /canyouhackit/
Disallow: /career-paths?title
Disallow: /salary-calculator-for-tech-hiring
Disallow: /daf/servlet/
Disallow: /jobs/dc-*
Disallow: /jobs/djt-*
Disallow: /products/?
Disallow: *?CMPID*
Disallow: */jobs.html
Allow: /jobs
Allow: /register
Allow: /mobile

User-agent: Googlebot-News
Disallow: /
```

Figure 2.3: Robots Exclusion Policy set by dice.com

Apart from these, many hosts regulate whats allowed/not-allowed for downloading by placing a *robots.txt* file under root directory of a hosted web site. The standard is known as Robot Exclusion Protocol. Figure 2.3 shows a *robots.txt* of dice.com. Its interpretation beginning at line one goes like no crawler should visit */admin*, */jobman* pages, but it can visit */jobs*, */register*, and so on. A user-agent containing string Googlebot-news is not legally allowed to crawl any of its pages. For each domain, the crawler fetches robots.txt to test whether the URL under consideration passes the robot restrictions and only then it is added to its Frontier data structure. Given a high locality that many of the extracted links fall within the domain, it is efficient to cache robots.txt rule into an in-memory data store. The cache expires and redownloads robots.txt for each domain several times a day to stay in sync with the rule changes.

It should be noted that it is always safer and courteous to include a note in request header of the crawler indicating intentions to download the data, and also provide your email address where the webmaster can contact you in case of any issues. Also, it is a good practice to enforce strict compliance with domain's robots.txt rules.

### 2.5.4 Duplicate URL Eliminator (DUE)

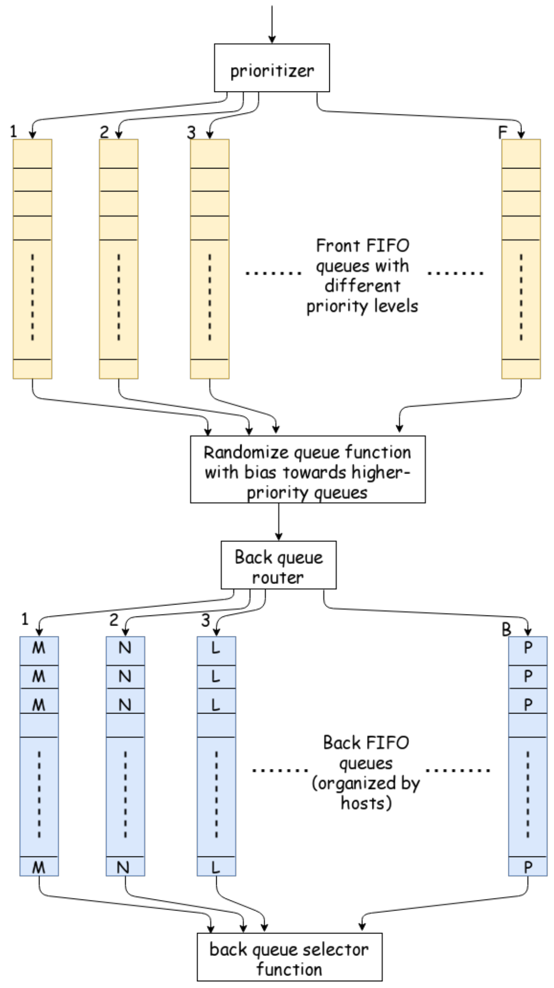
A batch of URLs qualified from URL filtering subsystem arrives at DUE. Sometimes it s also referred to URL-seen test. DUE guards the URL Frontier by eliminating multiple instances of the same URL from adding to the Frontier. It also keeps history of set of URLs that were previously added to the URL Frontier and those that are currently in it. The fire-and-forget, one-time crawl URLs are only crawled once. In continuous crawling Frontier scheme, some URLs are revisited periodically for new information, in such cases, the DUE has to delete its traces from its state.

Understanding the behavior of DUE explained in above paragraph, the size of URL set stored in relational database on disk will grow linearly as the size of web corpus grows irrespective of whether the crawler is continuous/non-continuous. Since the DUE has to make sure that it isn't adding duplicate URL to the Frontier, it will check against each entry in the URL set table. This will hamper DUE throughput and increase disk seek latency over time i.e the time taken to check and respond for existing entry increases. Each insertion in the URL set table is a fixed size checksum of textual URL along with the mapping to URL itself. The checksum algorithm should be such that it has exponentially small probabilistic bounds on the likelihood of collision in the URL set.

To reduce the number of operations required to DUE each URL in a batch, several optimizations can be combined. First, keeping an in-memory cache objects of popular URLs. The intuition for this is continuous crawling URLs endpoint will be revisited. But again with this, the cache hit rate is low as ratio of continuous to one-time crawls can be 20:1. Also as the computed checksums of the URLs are highly unique, it has lower locality, so the checksums that miss the cache will require disk access and disk seek. Secondly, taking advantage of high locality of checksum hostname(domain name) concatenating with checksum of absolute URL for that domain name results in same host names being numerically closer to each other. This in turn translates to access locality on the URL set.

### 2.5.5 URL Frontier

The URL Frontier subsystem receives a batch of URLs from its counterpart(it is either DUE or hostsplitter in case of Distributed crawling). It maintains those URLs in the Frontier and propagates them in some order whenever the fetch module seeks a URL from it.



prioritizer
1
2
3
F
Front FIFO queues with different priority levels
Randomize queue function with bias towards higher-priority queues
Back queue router
1
2
3
B
M
N
L
P
Back FIFO queues (organized by hosts)
back queue selector function

Figure 2.4: URL Frontier Scheme(based on Mercator)

Figure 2.4 is not just a simple queue but a complicated data structure. It includes multiple pages from the same host. It avoids trying to fetch all those pages from the same host at the same time because may be the crawler can overload the server which is not a good policy. Also, the fetch module should not wait doing nothing, instead keep the module busy. Thus, it balances the tradeoff between not bombarding the server and also keeping the fetch module busy so that if fetch is not crawling that website at least it can go to some other server and fetch some pages from there.

Following are important considerations that dictate the order in which the URLs are returned or pulled from the frontier. As mentioned earlier, simple priority queue fails as there is high locality of pages that go to its own site, creating a burst of accesses to that site.

- **Explicit politeness:** obeys any specifications from webmaster on what portions of the site site can be crawled, see section 2.5.3 for more details.
- **Implicit politeness:** Avoids hitting any site too often even though no specification exist on the site.
- **Freshness:** crawl some pages more often that others.

A common heuristic to handle politeness is to insert a time $t$ gap between successive requests $r$ such that,

$$t(r_i) > t(r_{i-1})$$
$$t(r_i) = (K.l(r_{i-1})) + TS_{now} \qquad \text{...}K \text{ is a variable delay factor}$$

It is important to visualize URL Frontier diagram figure 2.4 before diving into its pieces. A URL follows the path into the Frontier and immediately encounters set of front queues $F$. It enters into one of these $F$ queues. There are also another set of queues called $B$ back queues. At some stage the URL will be pulled out of the front queue and then sent into one these back queues. Later, at some stage, it will be pulled out of that back queue and accessed by fetcher thread which then goes and actually fetches the document at that URL endpoint.

Each front queue $F_m$ in range $\{1, 2, 3...m\}$ and back queue $B_n$ in range $\{1, 2, 3, ...n\}$ is literally a $FIFO$ queue. The incoming URL gets classified into one of the $1, 2, 3, ..F$ front queues

- **Front queues** $F_m$ manage prioritization. They influence the rate at which you ping the same web server again and again.
- **Back queues** $B_n$ enforces politeness.

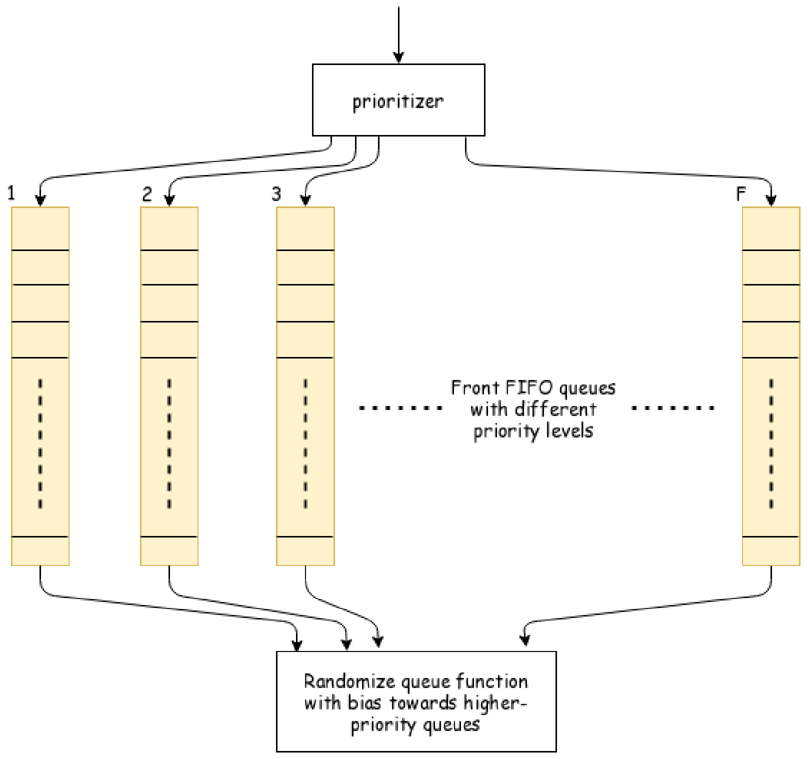

prioritizer
1
2
3
F
Front FIFO queues with different priority levels
Randomize queue function with bias towards higher-priority queues

Figure 2.5: Frontier Front Queue

Figure 2.5 gives a close inspection on Front Queues $F_m$. Incoming URL is assigned a integer priority between 1 & $m$. Based on the priority assigned it is going to enter that particular queue. priority $F_1$ holds URLs for web servers that need to be crawled very very frequently. For instance, let $L_s$ be links, $\forall L_s \in$ *dice.com* $\longmapsto F_1$ has highest crawling requirement whereas $\forall L_s \in$ *jobs.com* $\longmapsto F_m$ contains URLs which has least crawling requirement.

Heuristics for assigning priority,

- keeping track of how frequently the web pages are changing on a particular web server and how authoritative the pages are, if it occurs a case where a refresh rate is pretty high for any web server and pages happen to be important enough then those web servers are going to be crawled more often
- Application specific.

The prioritizer function for whirlpool is explained in chapter 3, subsection 3.3.5.1.Also randomize queue strategy on how the items get pulled out of this $F$ Front queues is explained in chapter 3, subsection 3.3.5.2. This sums the top half of the picture.

The back queue $B$ exist to achieve politeness because the crawler shouldn't hit the server too frequently. Again, there are $B$ back queues from $\{1, 2, 3, ...n\}$, so when the URLs are pulled out in biased order from $F$ Front queues, they are going to enter into one or more of these back queues.

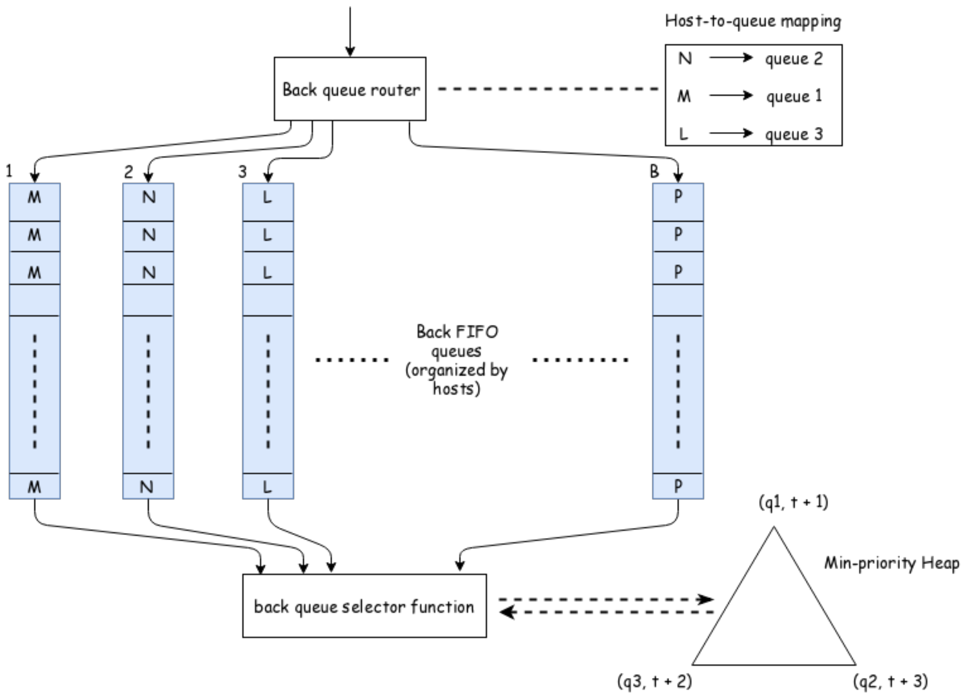


Figure 2.6: Frontier Back Queue

Recall the heuristic to handle politeness explained earlier, each back queue $B_n$ corresponds to a particular domain. For e.g back queue $B_1$ is for server *monster.com*, $B_2$ could be the server *indeed.com*, and so on. So all the URLs which have base name *dice.com*, when they emerge from the front queue enter queue $B_1$. Furthermore, Back queues maintains a minimum priority heap data structure in which the keys are such that the parent key $k$ is less than the key of both of its children i.e $k < 2k$ and $2k + 1$ and so the smallest item in the heap is at the root of the heap.

The size and vertex $V(q, t)$ of the heap is B. the key $q$ from vertex pair $(q, t)$ in it maps to one of the servers/back queues $B_n$. The value $t$ is going to be threshold timestamp $th_{ts}$ which is the value of time before which I am not allowed to hit the same server $B_n$ again. The very fact that back queue structure maintains a minimum heap is that its particular root represents the server which has the least value of the timestamp. Its the very server that is allowed to query next before it queries any of those other servers. A URL $u$ is pulled out from the head of server queue $q$ and fetched by fetch module. This is handled by back queue selector function

While this cycle continues, the back queue router ensures the server queues $B_n$ are non-empty. It is busy with pulling a URL $u'$ from the head of front queue picked up by some randomize front queue function. Following this, it checks whether server queue $q$ exists for $u'$, if so, then it gets added to that queue. It goes again to pull another URL $w$, this time it sees that server queue $q$ is now empty and there insert $w$ in it and renames the $q$ to $q'$. During the same iteration, the process sinks a new vertex $V'(q', t')$

on to the min-heap based on the meta-information of $w$ such as time $t'$.

### 2.5.6 Distributing Web Crawl

A crawler program as a whole can be distributed for various reasons. A distributed crawler can crank up crawling rate. This is achieved by making a identical clone of single node crawler and replicating it onto another machine. Another advantage of distributing the crawl is that a subset of websites are crawled by a machine which is geographically closer to its data center/region. The immediate question is how does mercator extend its architecture and also how do these nodes communicated and share URLs is shown in figure 2.7.

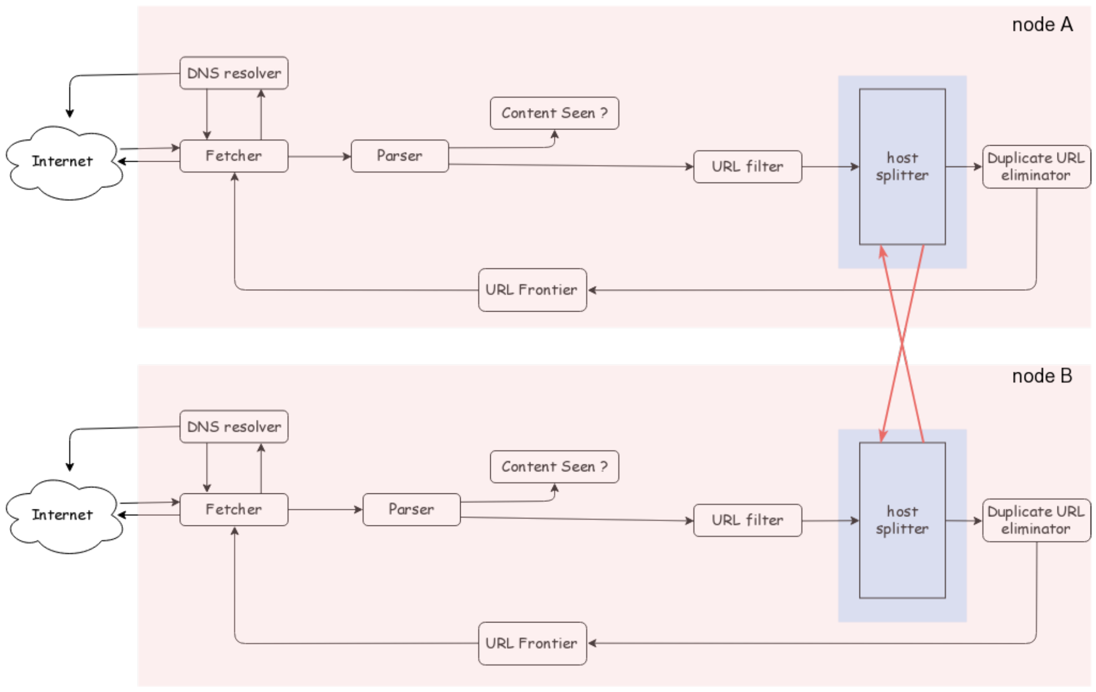


Figure 2.7: Host Partitioning

Lets say there are 3 different nodes designated to crawl the web $M_1, M_2, M_3$. All the hosts that are being crawled are partitioned into 3 sets. Imagine taking every single web server on the web or in the seed set, taking the URls for those servers, and hashing those URLs in the integer range $i \in \{1, 2, 3\}$. For instance, web server address *usjobs.com* could get hashed to an integer 2. So all those URLs that are being stored by that web server will be parsed by node $M_2$ while the other two nodes do not fetch documents located on *usjobs.com*.

The key to the implementation is the host splitting component shown in figure 2.7. The component at machine $M_1$ computes which machine is meant to crawl the URLs of *usjobs.com*. It takes the URL, looks at its server's address and hash it to the integer 2 upon which it knows $M_2$ is supposed to crawl URLs located on this server and therefore sends it to node $M_2$. Looking at the snapshot of node $M_2$, it has passed content seen, URL filter tests which confirms the incoming URL from different node is valid and need not go through the tests again. The only thing node $M_2$ has to check is whether that URL already exists in its own local URL frontier version or its own DUE. If the URL is new, the node will add it to its own URL frontier. At the same time node $M_1$ simultaneously receives URLs that it is supposed to crawl from those other nodes $\{M_2, M_3\}$.

## 2.6 Automata Analogy

## 2.7 Software Design Principles

### 2.7.1 Loose Coupling

### 2.7.2 DRY

### 2.7.3 Single Responsiblity

### 2.7.4 Open-Close Principle

### 2.7.5 Scaling the Design at various levels

## 2.8 Event-Driven Architecture

### 2.8.1 Message Queues (MQ)

### 2.8.2 MQ Advantages

### 2.8.3 RabbitMQ as a Message Broker

## 2.9 Amazon Web Services

### 2.9.1 Elastic Compute Cloud (EC2)

### 2.9.2 Virtual Private Cloud (VPC)

### 2.9.3 Relational Database Service (RDS)

### 2.9.4 Elastic Cache

### 2.9.5 MongoDB

### 2.9.6 Internet Gateway (IGW)

### 2.9.7 Network Address Translation (NAT)

### 2.9.8 AWS Monthly Calculator

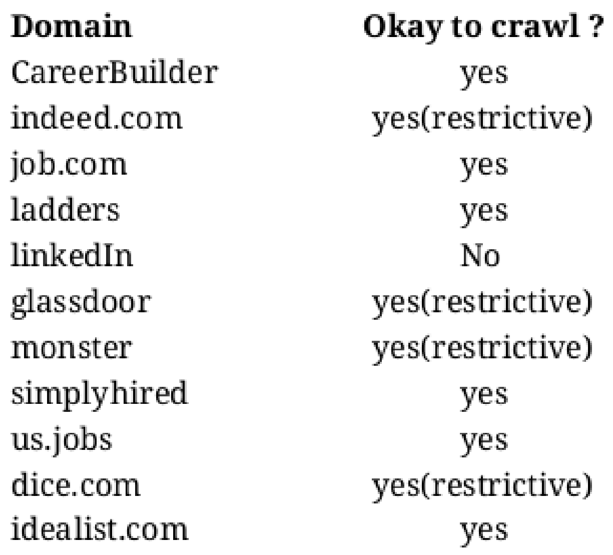
## 2.10 Docker Containers

### 2.10.1 DockerFile

### 2.10.2 Docker Compose

# 3 | Implementating Whirlpool

## 3.1 Mission



| Domain | Okay to crawl ? |
|---|---|
| CareerBuilder | yes |
| indeed.com | yes(restrictive) |
| job.com | yes |
| ladders | yes |
| linkedIn | No |
| glassdoor | yes(restrictive) |
| monster | yes(restrictive) |
| simplyhired | yes |
| us.jobs | yes |
| dice.com | yes(restrictive) |
| idealist.com | yes |

Figure 3.1: Topical crawling seed set

## 3.2 Characteristics

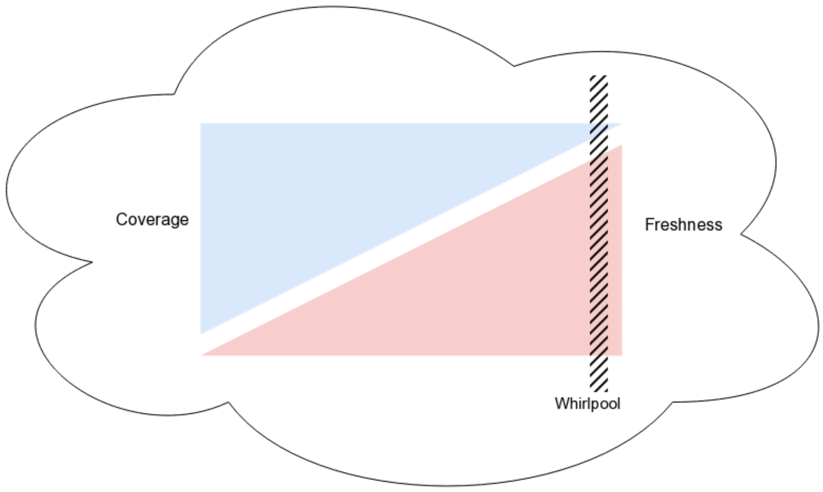


Figure 3.2: Crawl order policy of Whirlpool

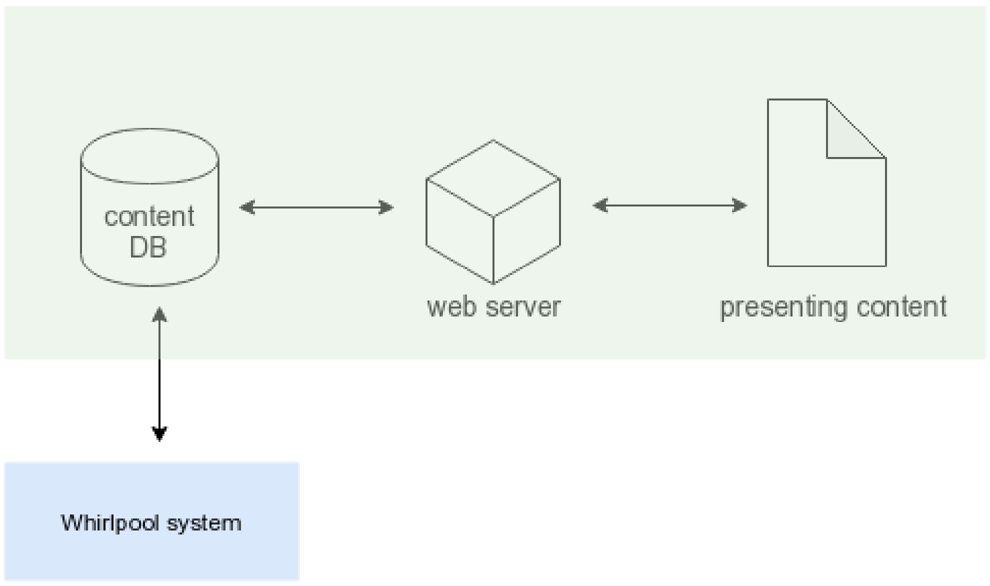


Figure 3.3: Presenting collected data

## 3.3 Internals

### 3.3.1 RabbitMQ as a Central Message Bus

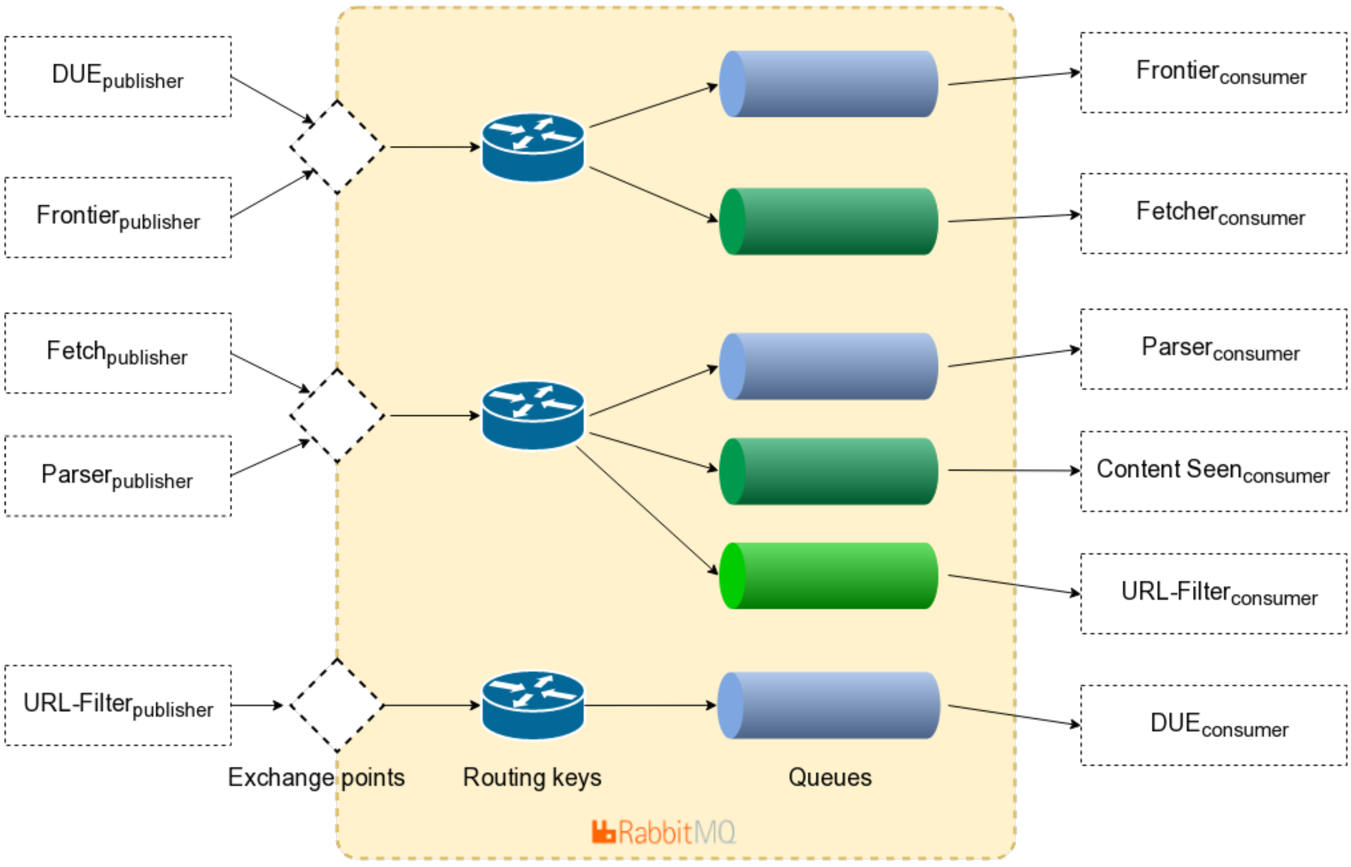


Figure 3.4: Message Distribution using RabbitMQ Direct Exchange

The initial rabbitmq configuration is shown in the https://github.com/rihbyne/whirlpool-rmq/blob/master/setup.j. The subsystems only need to know the exchange points and submit message payload to it. The exchange points take care of routing payload to its recipient queue. The payload is static, and doesn't contain any business logic, this leaves the subsystems highly decoupled from each other.

### 3.3.2 Shared Services with Docker

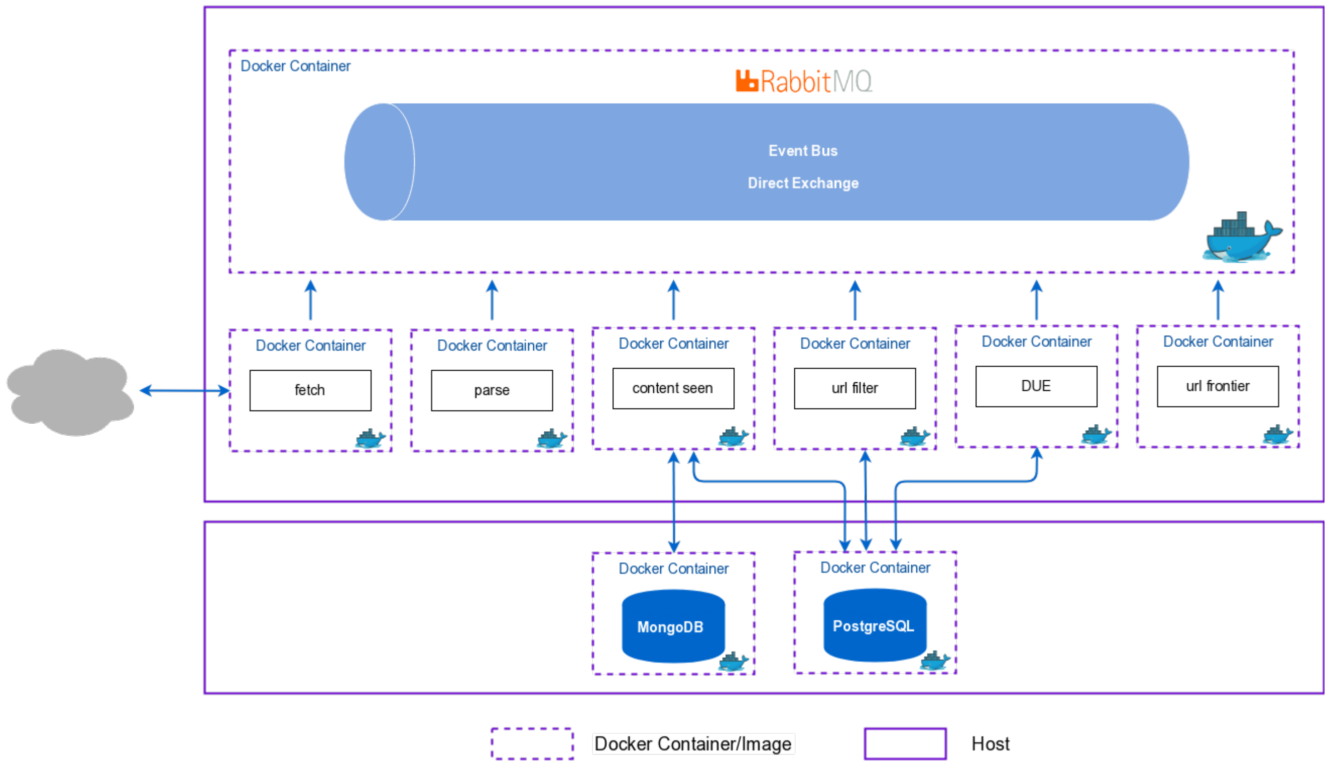


Figure 3.5: Single-node Whirlpool Subsystem Isolation using Docker

### 3.3.3 Parser & its Crawl Ordering Problem

### 3.3.4 URL filtering algorithm

### 3.3.5 URL Frontier

#### 3.3.5.1 Prioritizer Policy

#### 3.3.5.2 Front Queue Biased Selection Policy

### 3.3.6 Distributing the crawl(Linear vs. Consistent Hashing)

## 3.4 Boot Process

To be complete

# 4 | Deployment on AWS

This chapter discusses server farm assembly for running whirlpool on a on-demand hardware resources offered by AWS. It uses altitude analogy to illustrate important additions in infrastructure at various levels in detail. The last section explains hardware configuration of resources used and their pricing details.

## 4.1 Infrastructure from 10,000 feet

Given a working knowledge of AWS and a crawler which is replicated on to more than one machine, it becomes naturally obvious to setup a subnet within a Virtual Private Cloud(VPC) and launch EC2 instances in it. The subnet within a VPC belongs to single Availability Zone(AZ).

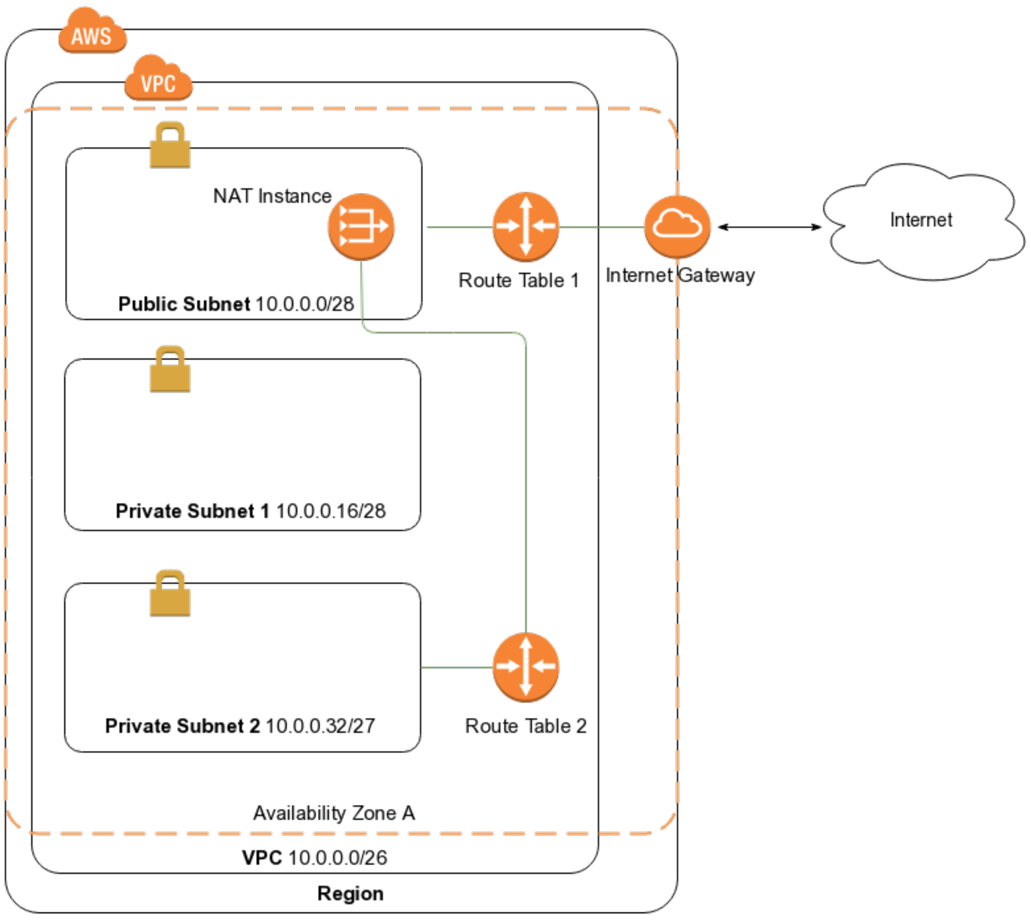


Figure 4.1: Whirlpool Infrastructure using VPC, AZ, and Subnet Sizing

Next, the data transfer in crawling process is such that it will make connection requests to the web sites and pull data into the amazon cloud, thus the inbound data transfer cost from the internet into amazon cloud is free. Security-wise, the crawler nodes should never except incoming connections from internet. These nodes are placed in private subnet 2. By default a private subnet within a AWS VPC is denied any inbound/outbound connections to the internet. As a best practice, subnet 2 is allowed only inbound connections. This is done by routing its internet traffic to NAT instance deployed in public subnet which further takes from Internet Gateway(IGW) attached to the VPC all the way into public internet.

Moreover, the subsystems of whirlpool leverage relational database to maintain a history of URLs, NoSQL to store extracted text which is then used by the presentation tier to render the data to the public. Also, later on, a cache server can be adopted to optimize lookup efficiency of various components. Given this requirement, its safer to form a second private subnet dedicated to placement of data stores and a third subnet which is public to host application server as shown in the figure 4.1. This way, the data store subnet would stay safer accepting network connections only within the VPC.

Figure 4.1 uses Classless Inter-Domain Routing(CIDR) block of $10.0.0.0/26$ which gives a range of $64$ IPv4 addresses. This thesis uses a tool [17] to calculate CIDR blocks. AWS VPC reserves first four and last IPv4 addresses of each subnet created for internal usage and therefore cannot be assigned to any instance of any resource. The primary CIDR block size of VPC is used then used to create 3 subnets within the VPC each with a non-overlapping CIDR block size. The minimum subnet size allowed is $28$ and $16$.

CIDR block size is given by $2^{(32-x)}$, substituting $x = 26$ yields,

$$2^{(32-26)} = 64 \qquad \text{IP addresses}$$

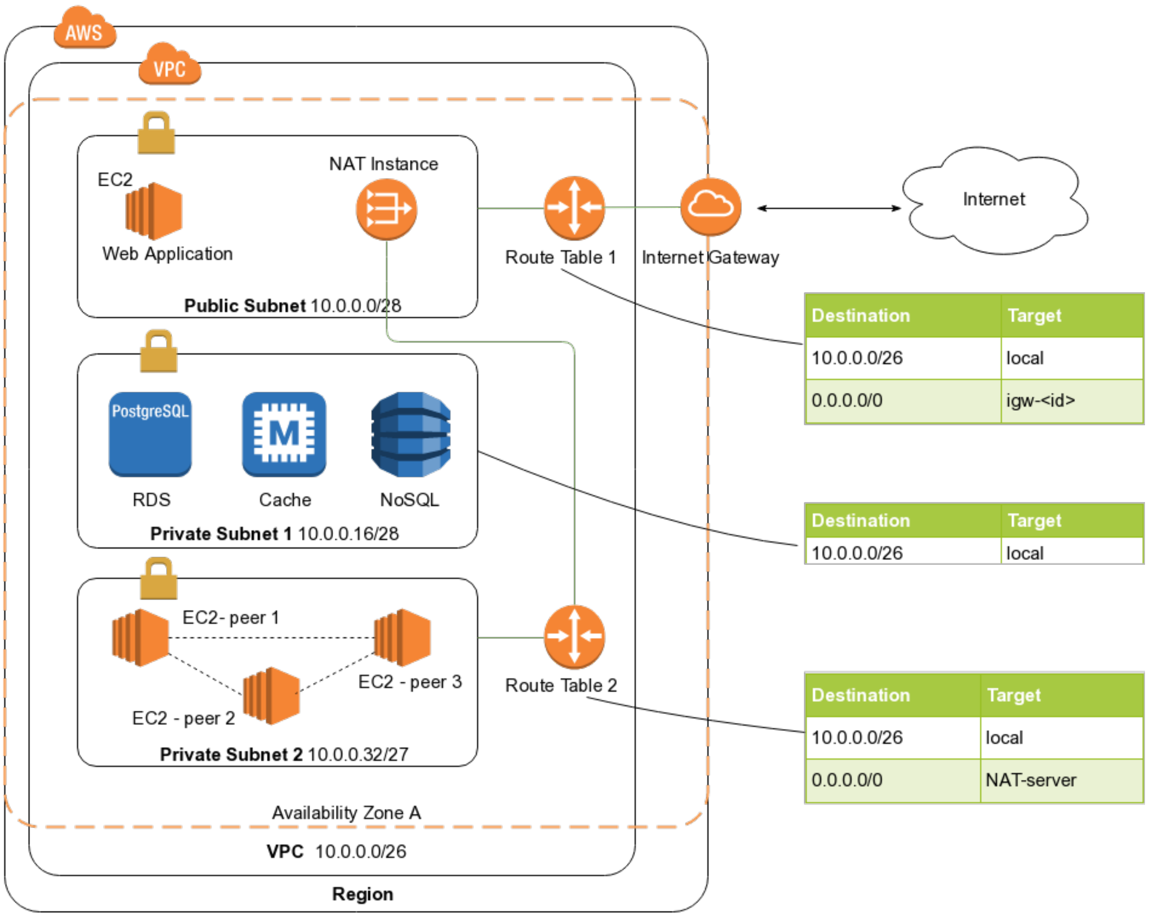
3 subnets formation = $\underbrace{16}_{\text{public}} + \underbrace{16}_{\text{1st private}} + \underbrace{32}_{\text{2nd private}}$

Actual IP addresses available = $\underbrace{11}_{\text{public}} + \underbrace{11}_{\text{1st private}} + \underbrace{27}_{\text{2nd private}}$

Public subnet IP range = $10.0.0.0/28 \rightarrow$ (0 - 15)
1st Private subnet IP range = $10.0.0.0/28 \rightarrow$ (16 - 31)
2nd Private subnet IP range = $10.0.0.0/28 \rightarrow$ (32 - 64)

## 4.2 Infrastructure from 5,000 feet

Now, following up from section 4.1, the 2nd private subnet is the place where the crawler resides.



| Destination | Target |
|---|---|
| 10.0.0.0/26 | local |
| 0.0.0.0/0 | igw-<id> |

| Destination | Target |
|---|---|
| 10.0.0.0/26 | local |

| Destination | Target |
|---|---|
| 10.0.0.0/26 | local |
| 0.0.0.0/0 | NAT-server |

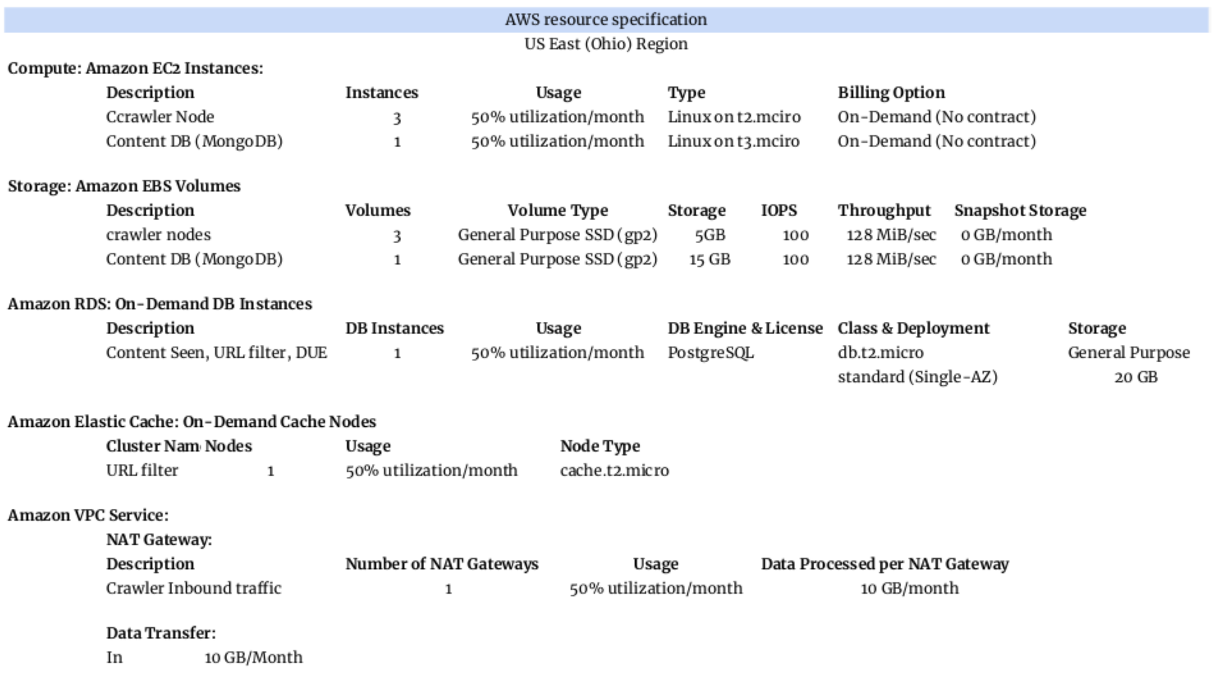
Figure 4.2: Whirlpool infrastructure with Route tables, NAT

Each subnet is assigned a default route tables as depicted in figure 4.2. NAT instance in public subnet forwards traffic from instances in subnet 2 to the internet and and send the response for corresponding request back to those instances in subnet 2. It won't allow outside clients to initiate connections with instances in subnet 2. The public subnet consists of one application server(EC2) and a NAT instance. The custom route(shown in route table 2) is created and attached to crawler subnet. The custom rule directs the traffic originated within any of the private subnet peers matching subnet mask $0.0.0.0/0$ to NAT server.

At this stage, the public subnet is not really public unless a Internet Gateway(IGW) is attached. After creating a Internet Gateway(IGW), a custom route(show in route table 1) is created and attached to the public subnet. This will scope all packets matching $0.0.0.0/0$ route to IGW. The traffic from private crawler subnet will flow to NAT Instance and then to the IGW. The NAT will translate back-and-forth source and destination IPs of private instances.

The data store private subnet contains RDS instance(PostgreSQL as part of the implementation). The extracted data is persisted in MongoDB which is a NoSQL variant. AWS does not have managed instance of MongoDB and requires the interested party to operate, maintain on an EC2 Instance. Finally, to improve lookup time efficiency of few subsystems within the whirlpool, AWS ElasticCache is leveraged.

## 4.3 AWS Services Hardware Specfication & Cost Estimation



AWS resource specification

US East (Ohio) Region

**Compute: Amazon EC2 Instances:**

| Description | Instances | Usage | Type | Billing Option |
|---|---|---|---|---|
| Ccrawler Node | 3 | 50% utilization/month | Linux on t2.mciro | On-Demand (No contract) |
| Content DB (MongoDB) | 1 | 50% utilization/month | Linux on t3.mciro | On-Demand (No contract) |

**Storage: Amazon EBS Volumes**

| Description | Volumes | Volume Type | Storage | IOPS | Throughput | Snapshot Storage |
|---|---|---|---|---|---|---|
| crawler nodes | 3 | General Purpose SSD (gp2) | 5GB | 100 | 128 MiB/sec | 0 GB/month |
| Content DB (MongoDB) | 1 | General Purpose SSD (gp2) | 15 GB | 100 | 128 MiB/sec | 0 GB/month |

**Amazon RDS: On-Demand DB Instances**

| Description | DB Instances | Usage | DB Engine & License | Class & Deployment | Storage |
|---|---|---|---|---|---|
| Content Seen, URL filter, DUE | 1 | 50% utilization/month | PostgreSQL | db.t2.micro standard (Single-AZ) | General Purpose 20 GB |

**Amazon Elastic Cache: On-Demand Cache Nodes**

| Cluster Nam | Nodes | Usage | Node Type |
|---|---|---|---|
| URL filter | 1 | 50% utilization/month | cache.t2.micro |

**Amazon VPC Service:**

**NAT Gateway:**

| Description | Number of NAT Gateways | Usage | Data Processed per NAT Gateway |
|---|---|---|---|
| Crawler Inbound traffic | 1 | 50% utilization/month | 10 GB/month |

**Data Transfer:**

| | |
|---|---|
| In | 10 GB/Month |

Figure 4.3: Whirlpool Hardware Specification on AWS

From table 4.3 under Amazon EC2 services, a crawler node sits on a on-demand EC2 instance which is of type t2.micro. The t2.micro has single core virtual CPU (vCPU) with 1 GiB of Memory. Each crawler node runs a message broker - RabbitMQ to interconnect the crawler subsystems. According to RabbitMQ Documentation [6], the host should have at least 128 MB memory available at all times. Moreover, it wont accept any new messages when it detects that its using more than 40% of avialable memory. Being said that, there is always an option to change the current instance type from t2.micro to t2.small and so on depending on usage. MongoDB which collects extracted text is hosted on On-Demand t3.micro instance type, which is a 1 GB, dual-core vCPU operating at 50% of its capacity.

Coming to Amazon Storage, the crawler node is coupled with Elastic Block Storage(EBS) volume of 5 GB. The volume is general purpose SSD capped at 100 IOPS giving a throughput of 128 MB/sec. For mongoDB an EBS volume of 15GB is used. It is formatted with XFS filesystem for its data directory as recommended in its production checklist.

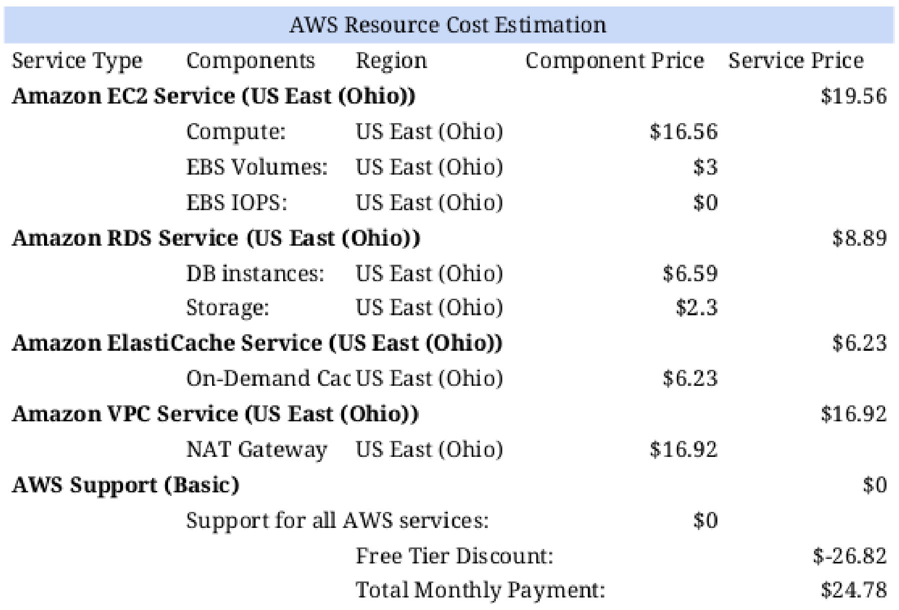
Whirlpool uses one On-Demand RDS instance with PostgreSQL engine operating at

50% utilization monthly. The underline managed Operating System is a db.t2.micro deployed in a single AZ. PostgreSQL to store fingerprints of a web page content at a given URL, robots.txt attributes of each site, and URLs already enqueued to be crawled.

Amazon ElasticCache provides a choice between Memcached & Redis Instance. It will be used by URL filter subsystem of whirlpool.

Lastly, Amazon VPC has a NAT Gateway which will process data estimated close to 10 GB/month. Regardless of data flow inward/outward, the NAT is charged by the hour.

Table 4.4 provides approximate monthly billing information of AWS services used by this project to build, test, and run experiments. The calculation was performed using AWS monthly calculator. At the time of this writing, the author is enrolled is 12-month AWS Free tier access which discounts most of the services the crawler system leverages.



| AWS Resource Cost Estimation | | | | |
|---|---|---|---|---|
| Service Type | Components | Region | Component Price | Service Price |
| **Amazon EC2 Service (US East (Ohio))** | | | | $19.56 |
| | Compute: | US East (Ohio) | $16.56 | |
| | EBS Volumes: | US East (Ohio) | $3 | |
| | EBS IOPS: | US East (Ohio) | $0 | |
| **Amazon RDS Service (US East (Ohio))** | | | | $8.89 |
| | DB instances: | US East (Ohio) | $6.59 | |
| | Storage: | US East (Ohio) | $2.3 | |
| **Amazon ElastiCache Service (US East (Ohio))** | | | | $6.23 |
| | On-Demand Cac | US East (Ohio) | $6.23 | |
| **Amazon VPC Service (US East (Ohio))** | | | | $16.92 |
| | NAT Gateway | US East (Ohio) | $16.92 | |
| **AWS Support (Basic)** | | | | $0 |
| | Support for all AWS services: | | $0 | |
| | | Free Tier Discount: | | $-26.82 |
| | | Total Monthly Payment: | | $24.78 |

Figure 4.4: Whirlpool Hardware Cost Estimation on AWS

With the free-tier in use, the total price is dropped almost by 50 %. Under free tier, EC2 is limited to 750 hours/month which equates to approx. 24 hours/day for 30 days using only Linux, RHEL. Any combination of EBS (SSD/Magnetic) is 30 GiB. Amazon RDS again limited to 750 hours/month upto 20GB of general purpose SSD database storage. Amazon ElasticCache - 750 hours. NAT Gateway is charged $ 0.045 an hour. There is no charge for data transfer between cross-region multi-AZ as this implementation is 1 region, single AZ. Also, the data transfer into AWS cloud is free.

# 5 | Experiment Results & Justification

This section depends on lessons learned and any observations made in section 3. Yet to be completed

## 5.1 Whirlpool as a Microserivce Application

To be completed

# 6 | Conclusion & Future Work

This chapter is a list of follow-up ideas that can be turned into potential projects. These are complex and important topics that deserve a report of their own. This paper wont wouldn't do them justice by making them superficial side notes. The ideas are grouped by subject of interest.

## 6.1 Systems

### 6.1.1 Experimenting with Distributed Data

## 6.2 Search

### 6.2.1 Data Indexing using ElasticSearch Engine

## 6.3 Frontend

### 6.3.1 GUI Admin & Monitoring Dashboard

## 6.4 Machine Learning (ML)

### 6.4.1 Discovering new relevant websites by adding Comprehensive Coverage using ML techniques

### 6.4.2 Categorizing Jobs using Supervised/RNN classifier

# 7 | Timeline

I intend to graduate in fall 2019 i.e next semester. Below is my timeline to execute the thesis starting first week of June, 2019:

(Week 1 - 7) First get the single node crawler running on local development environment. To achieve this,

- (Week 1) Build docker images of data stores involved. I will be using PostgreSQL for fingerprintDB, Robots Exclusion Policy, and DUE url set. The job data will be stored in MongoDB(development & production). Each data store will maintain local.yml & production.yml version of dockerfile which will point to local and aws version of databases, respectively.
- (Week 1) Build docker image of RabbitMQ with default configuration. Followed by getting acquainted with enough terminology and tutorials to get started. This image will be shared with all subsystems of a crawler.
- (Week 2) Initialize git repository for each subsystem of the crawler. Build docker image for each subsystem, specify RabbitMQ as its dependency. Bind each subsystem to a queue through a message broker depending on its role as a consumer/producer. Test for open connections of each consumer/producer.
- (Week 2) Boot all the subsystems in sequence using docker compose. Test the message passing between services. This stage ensures the tooling required is up & running as expected.
- (Week 3) URL Frontier queue is the 1st step of the crawler subsystem. Initially, maintain a single FIFO queue under the frontier with seed sets listed in section 3.1. Code functionality for consumer & producer scripts bound to a queue of fetcher & DNS subsystem.
- (Week 4) Given the seed URL $u$ and a page $p$, code functionality for consumer & producer scripts of Parser, Content Seen subsystems, respectively as separate services. Test message passing among fetcher, parser , & content seen.
- (Week 5) Next, once we have extracted links from page $p$, code functionality for consumer & producer scripts bound to a queue for URL filter subsystem. Test message passing between this filter and previous subsystems implemented.
- (Week 6) DUE tracks present & past history of URLs in Frontier regardless of whether they are 1-time crawl or continuous crawls. Implement consumer & producer scripts for DUE. Test message passing between subsystems implemented so far.
- (Week 7) Write code for front(priority) and back(politeness) queue of URL Frontier subsystem. Verify messages published & consumed. Test message passing between subsystems implemented so far.

| | |
|---|---|
| (Week 8) | At this stage, ensure the crawler as a whole is running as a single node on local machine. |
| (Week 8 & 9) | Next, Setup AWS Infrastructure for whirlpool by following the diagram. Build and test the docker images in isolation pointing to AWS data stores within the pvt subnets. |
| (Week 10) | Get crawler running on single node within a private subnet of AWS VPC. |
| (Week 10 & 11) | Initialize a repo. for host splitting module. Implement consistent hashing algorithm. This will parallelize crawling. |
| (Week 11 & 12) | Make 2nd identical clone of the crawler within a private subnet of AWS VPC. Observe & verify traffic is split equally between the nodes. |
| (week 13) | Demonstrate Whirlpool working to Dr. Soltys. Make any changes requested in the code or the report. |

# References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.*

[2] Martin Kleppmann (2016). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*

[3] Jeffrey Barr (2010). *Host Your Web Site in the Cloud: Amazon Web Services Made Easy*

[4] Kyle Banker (2011). *MongoDB in Action*

[5] Martin Abbott, Michael Fisher (2011). *Scalability Rules: 50 Principles for Scaling Web Sites*

[6] Alvaro Videla and Jason J. W. Williams. (2012) *RabbitMQ in Action: Distributed Messaging for Everyone*

[7] Olston, C., & Najork, M. (2010). *Web crawling. Foundations and Trends in Information Retrieval, 4(3), 175-246.*

[8] Heydon, A., & Najork, M. (1999). *Mercator: A scalable, extensible web crawler.World Wide Web, 2(4), 219-229.*

[9] Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2004). *Ubicrawler: A scalable fully distributed web crawler. Software: Practice and Experience, 34(8), 711-726.*

[10] Lee, H. T., Leonard, D., Wang, X., & Loguinov, D. (2009). *IRLbot: scaling to 6 billion pages and beyond. ACM Transactions on the Web (TWEB), 3(3), 8.*

[11] Harth, A., Umbrich, J., & Decker, S. (2006). *Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In The Semantic Web-ISWC 2006 (pp. 258-271). Springer Berlin Heidelberg.*

[12] Seeger, M. (2010). *Building blocks of a scalable web crawler. Master's thesis, Stuttgart Media University.*

[13] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997) *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*

[14] https://www.docker.com/

[15] https://microservices.io/

[16] https://aws.amazon.com/

[17] http://www.subnet-calculator.com/cidr.php

[18] https://www.docker.com/

[19] https://archive.org

[20] https://www.softwareheritage.org/