Improvised Architecture for Distributed Web Crawling

Tilak Patidar
Computer Science & Engineering,
SRM University, Chennai

Aditya Ambasth
Computer Science & Engineering,
SRM University, Chennai

ABSTRACT
Web crawlers are program, designed to fetch web pages for information retrieval system. Crawlers facilitate this process by following hyperlinks in web pages to automatically download new or update existing web pages in the repository. A web crawler interacts with millions of hosts, fetches millions of page per second and updates these pages into a database, creating a need for maintaining I/O performance, network resources within OS limit, which are essential in order to achieve high performance at a reasonable cost. This paper aims to showcase efficient techniques to develop a scalable web crawling system, addressing challenges which deals with issues related to the structure of the web, distributed computing, job scheduling, spider traps, canonicalizing URLs and inconsistent data formats on the web. A brief discussion on new web crawler architecture is done in this paper.

Keywords
Web Crawler, Distributed Computing, Bloom Filter, Batch Crawling, Selection Policy, Politeness Policy.

1. INTRODUCTION
Web crawlers are programs that exploit the graph structure of the web moving from page to page. More commonly, related to web imagery they are also known as wanderers, robots, spiders, fish and worms. The key reason behind developing crawlers is to create a knowledge repository from these pages enabling user to fetch information from a search engine. In its simplest form a crawler starts from a seed page and then uses the links within it to advance to other pages acquiring more links associated with it, until a sufficient number of pages are identified or no further links exists. Although, the problem stated seems quite easy but behind its simple description lies a host of issues related to network connections, I/O and OS limits, spider traps, obtaining fresh content, canonicalizing URLs, inconsistent data schemas, dynamic pages and the ethics of dealing with remote web servers. As Shkapenyuk and Suel [1] shrewdly noted that:

"While it is fairly easy to build a slow crawler that downloads a few pages per second for a short period of time, building a high-performance system that can download hundreds of millions of pages over several weeks presents a number of challenges in system design, I/O and network efficiency, and robustness and manageability."

Today, web crawlers are not just a program maintaining a queue of pages to be crawled but they have evolved and combined with various agile services to form integrated high scale distributed software.

One of the common misconceptions observed is that once a particular domain is crawled, further re-crawl is not required. However, the web is dynamic and many new pages are added every second. To get new published pages in crawl cycle one can use various available resources like sitemaps, RSS feeds etc. which shall be discussed below.

A good crawler must be good at two things [1]. First, it should demonstrate a good strategy for deciding which pages to download next. Secondly it must fetch large amount of pages in short span of time while expressing robustness against crashes, while being manageable and considerate of resources and web servers.

There are various studies on strategies for crawling important pages first [2][3], crawling pages based on topic, or re-crawling pages in order to provide freshness of index. So put together the behaviour of a web crawler is the outcome of combination of the following policies [4]. A selection policy that states which pages to download, a re-visit policy that states when to check for changes to the pages, a politeness policy that states how to avoid overloading websites, and a parallelization policy that states how to coordinate distributed web crawlers.

2. LITERATURE SURVEY
Web crawler had been realized in the year 1993. Matthew Gray implemented the World Wide Web Wanderer [4]. The Wanderer was written in Perl and ran on a single machine. Three more crawler-based Internet Search engines came in existence: Jump-Station, the WWW Worm and the RBSE spider.

Though, most of these Web Crawlers used a central crawl manager which manages parallel downloading of web pages. Their design did not focus on scalability, and several of them were not designed to be used with distributed database management systems to which are required for storing high volume of links and unstructured data from web pages.

Internet Archive crawler was the first paper that addressed the challenges of scaling a Web Crawler. It was designed to crawl on the order of 100 million URLs. Hence it became impossible to maintain all the required data in main memory. The solution proposed was to crawl on a site-by-site basis, and to partition the data structures accordingly. The IA design made it very easy to throttle requests to a given host, thereby addressing politeness concerns, and DNS and robot exclusion lookups for a given web to be crawled in a single round. However, it is not clear whether the batch process of integrating off-site links into the per-site queues would scale to substantially larger web crawls.

Another famous crawler is Apache Nutch which runs on Hadoop ecosystem. It is written in Java and is used alongside with Elasticsearch or Apache Solr for indexing crawled data. Nutch runs completely as a small number of Hadoop MapReduce jobs that delegate most of the core work of fetching pages, filtering and normalizing URLs and parsing responses to plug-ins. It supports distributed operation and therefore be suitable for very large crawls. However there are some drawbacks to Nutch. The URLs that Nutch fetches is determined ahead of time. This means that while you're fetching documents, it won't discover new URLs and immediately fetch them within the same job. Instead after the fetch job is complete, you run a parse job, extract the URLs,

add them to the crawl database and then generate a new batch of URLs to crawl. The re-crawl time interval is same for all the domains in the seed file irrespective of domain demand (for e.g. News websites requires daily crawl). With all these shortcomings scaling beyond 100 million pages is still not achievable [5].

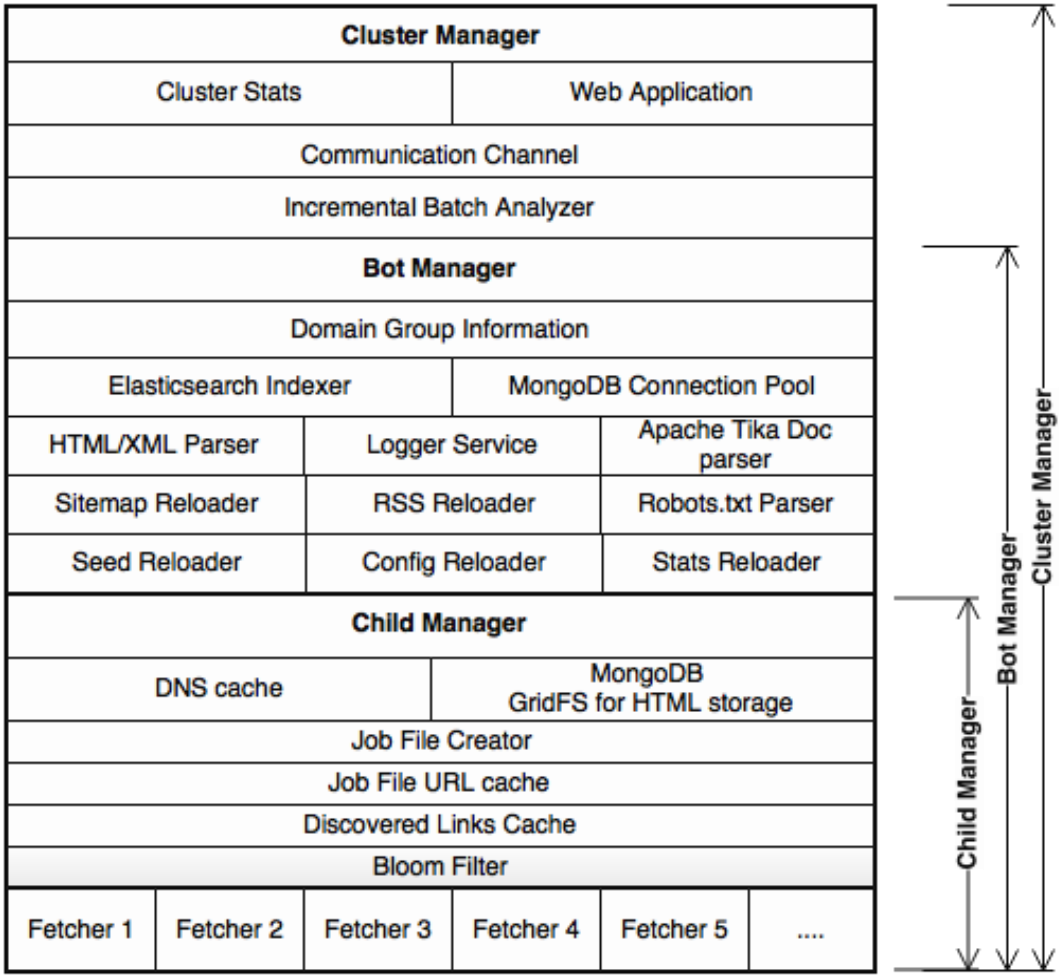

Fig 1: Component diagram for proposed crawler

3. PROPOSED WORK

3.1 Components of proposed web crawler

Fig.1 suggests the block diagram for the overall component in the proposed crawler. The three main components here viz. child manager, bot manager and cluster manager present in the proposed hierarchical structure. The cluster manager is responsible for bot-to-bot communication, cluster statistics and also provides a web application for managing bots within the cluster. The bot manager stands as the backbone layer to the web crawler. It includes the essential services for crawling web pages. The child manager component manages and coordinates the activities between various spawned fetcher processes set by the bot manager.

3.1.1 Child manager

The child manager fetches new batches from the central queue which is present in the MongoDB stack. Each job file allocated comprises of set of links to be fetched. It also contains necessary information such as associated domain group i.e. to which group those domains belong and the next timestamp for the re-crawl. The same is explained with the help of fig.2

If certain web pages are failed while crawling, they are reported to the child manager which then adds these failed web pages to a 'fail queue'. Each failed page is retried at least thrice before marking it as failed page. Here after the regex filtered links and the parsed HTML content obtained are hashed using MD5 and then these are checked for de-duplication in bloom filter [6][7]. The De-Duplication test is carried out in the bloom filter. The same is explained with the help of fig. 3

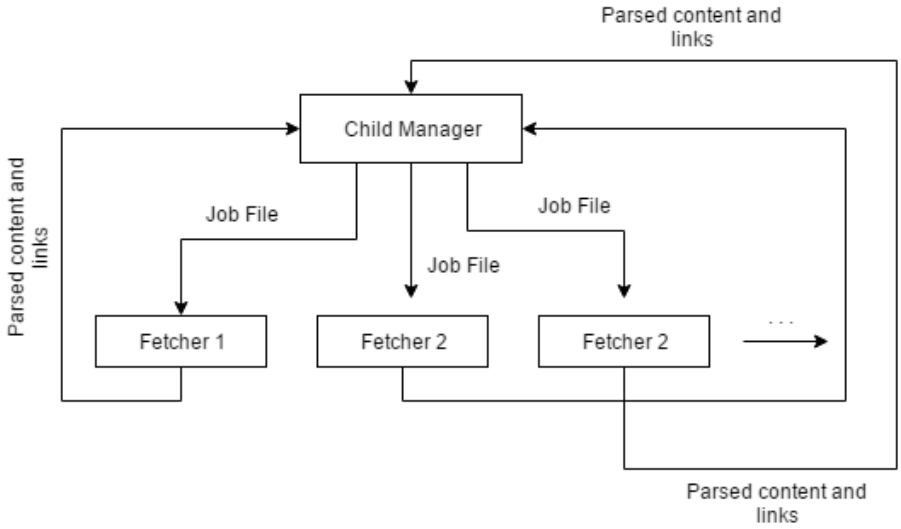

Fig 2: Job allocation to fetcher processes in Child Manager

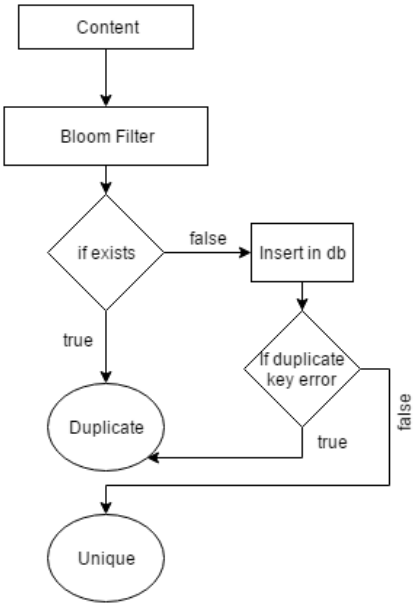

Fig 3: Content duplication test using bloom filter

The following is the configuration of bloom filter used:

- n = 10,000,000 no of unique URLs expecting => 1 million
- m = 287,551,752 (34.28MB) size of bloom filter in bits
- k = (n/m) ln(2) number of hash function to use ~= 20

Probability of false positives for this bloom filter is 1.0E-6 (1 in 1,000,000)

There are two major disadvantages of completely relying on bloom filters [7]:-

- False positives - can be improved by optimizing values of n, m and k
- Aging of bloom - This happens when bloom filter is filled, results in increase of false positives. To deal with this issue clear bloom filter can be cleared once it reaches a decided size.

So, the proposed crawler does not completely rely on bloom filters for detecting URL duplicates and content duplicates. Bloom filter acts as the first test and database insertion technique as a second test to eliminate duplicate content.

After the De-Duplication test, the filtered links are added into the discovered links cache from where the links are batch inserted into the database. The URLs which are not assigned to a job file are fetched and are inserted into job file URL cache from where they are selected for job creation process. Likewise the job file creator fetches links from this cache based on given domain, sorted by the absolute depth of the URL path. Thus this kind of sorted fetch gives priority to the URL belonging to the top breadth. The job files are assigned a cumulative score in inverse order of URL directory length.

DNS cache is used to avoid thrashing of local DNS server and reduce the response time because DNS lookups generate significant network traffic [1]. The fetched HTML files are stored for future usage in MongoDB's grid file system which provides distributed storage and helps in managing billions of text files available on the storage media.

3.1.2 Bot manager

The backbone of crawler i.e. bot manager holds small but essential services. With engrossed services to monitor, certain events such as configuration change, new added domains, RSS feed changes and sitemap changes along with bot-status update, on event trigger. Hence these are used to update the status of the bot.

The Bot manager is also responsible for providing document parsing services. So HTML/XML parser and Apache Tika Doc parser are incorporated to supported varied document formats.

3.1.3 Cluster manager

The cluster manager is responsible for coordination between multiple bots deployed within the cluster. A communication channel is established between the bot, utilized by the web application for providing the real time monitoring and configuration management.

3.1.3.1 Incremental batch analyser

For efficient re-crawling, the proposed crawler analyses the amount of change in the web-pages at every re-crawl. Using parameters such as content update from RSS feeds, tracking changes in sitemap index and content change over a fixed

period, give us the average refresh interval for the web page. To optimize batches with respect to refresh interval a component is introduced stated as incremental batch analyser. This aligns with the idea of incremental crawling [8] where it calculates and uses average refresh interval and hence, regroup URLs for better re-crawls.

3.2 Selection policy

3.2.1 Breadth-First as a selection policy

For selecting important pages many known techniques like breadth first, page rank, backlink count etc. are suggested. However, page rank gives most optimum selection, but its computation takes space as well as time [3]. When crawl queue is big, calculating page rank will slow down other operations [3]. Also, the difference between optimum crawls using page rank and breadth first is not very significant [2]. Breadth first selection scheme is used in the proposed web crawler architecture.

Performing breadth first implies to be easy on a single node crawler using a queue. This queue can be implemented using a SQL table or ordered NoSQL collection. However, implementing a breadth first in a distributed environment is difficult.

3.2.2 Single URL selection vs. Batch selection

In web crawling main focus is to provide more fetching cycles and managing I/O resources efficiently. In this case selecting one URL at a time for crawling cycle seems inefficient. Many URL selection policies have been discussed for optimum crawling [2]. But they fail to realize that a high scale distributed crawler cannot wait to acquire a single URL from a queue (database). Thereby waiting for every database call which would result in millions of hung threads in a large scale distributed environment. To eliminate this issue the proposed crawler is not selecting URLs, but batches of URLs [4]. Many high scale crawlers like Apache Nutch etc. realize the I/O problems in fetching single URL for every crawl. However, they follow sequential methodology for generating batch(s) of URLs which require entire URL table scan. The batch generation system in the proposed crawler eliminates the requirement for table scans and uses in-memory cache to achieve fast and parallel batch(s) creation.

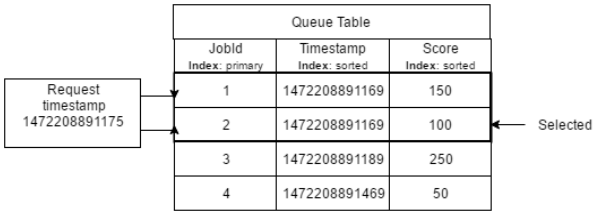

Fig 4: Selection of job file based on request time-stamp

The batch of URL is referred as a job file. A job file holds two essential properties i.e. timestamp and score. Timestamp is the time for job file to be re-crawled. Score is calculated on the basis URL quality and uniform distribution of URLs. So Fig.4 depicts that when a fetcher process requests for next batch of URLs, so the job file is selected having lesser timestamp than the request timestamp. In case of multiple jobs satisfies the criteria, is to sort by score.

3.2.3 Updating new content

Much research has been done on re-crawling or refreshing the search index [8] [9]. To do so without fetching the entire website, one has to monitor external changes like search trends, traffic logs, etc. Re-crawling based on topic [8] will first require doing topic modelling on crawled data [10]. Sitemaps and RSS feeds [11] in the website contain this updated information. However sitemaps are updated once a day, RSS feeds are almost updated every minute.

So in the proposed crawler, sitemaps and RSS feed parsing runs all time in background to check for new content. The RSS links found when parsing the HTML content are inserted into RSS links collection. Additionally, the crawler utilizes RSS feeds based on tags of discussion which are provided by various online retail websites, blogs and forums. For e.g. reddit provides RSS feeds for each discussion tag. Also, tracking social media pages belonging to a webpage, yields latest content.

3.3 Politeness policy

Politeness policy refers to not overwhelming a particular web domain with concurrent requests. It demands separating URLs by web domain with sufficient time interval to avoid banishment of the crawler. There has been a lot of discussion on this topic. Sometimes crawlers deliberately reduce the number of requests or add sleep time when large number of requests has been made [1]. So, to define standards there are two approaches for making crawler polite while crawling. First, is to reduce the number of requests or introduce sleep time in threads. Second, is to distribute the crawling load on various websites. Thus, still crawling millions of pages per second but not being blocked because of distributed number of requests on various web sites.

The second method is chosen, so that millions of http requests are made without getting blocked. Thus, many fetcher jobs complete in one cycle. Contrary to introducing sleep time in threads as suggested in the first method.

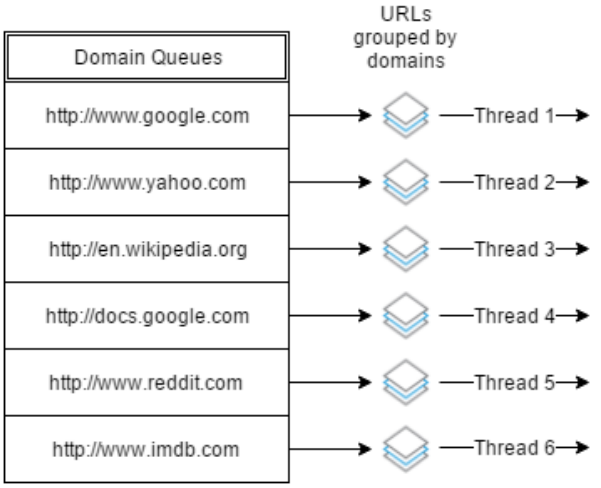

Fig 5: Creation of domain queues for politeness

The URLs submitted to the fetcher are grouped by domain. Thus, creating multiple domain queues as shown in Fig.5. The robots.txt parser in the bot manager provides the number of requests that a crawler can make to a specific domain. Based on this URLs are de-queued from each domain queue and HTTP connections are open in new threads. As, soon as URLs from one website is finished it is replaced by another URL from the queue until, all the queues exhaust. To give optimum performance the queues must contain nearly same number of URLs. Thus, terminates the fetcher threads within same duration. Also the uniform distribution of URLs in job file results in similar length domain queues which increases fetcher performance.

3.3.1 Seed file partitioning

Web crawlers suffer performance issues when the seed file is big (about 100 thousands of domains). Here withholding these big seed files in the memory gets difficult.

To, handle this issue seed file partitioning, is chosen as first layer of distribution. In this partitioning scheme the domains having same fetch interval are grouped together. Each domain group is assigned to a bot. It can have multiple domain groups. The advantages observed are mentioned below:-

- A bot does not need to know what other bots are crawling. (Isolation)
- It ensures that a particular domain group is crawled by only one bot at a time. (Politeness)
- Grouping domains based on fetch interval helps in scheduling based on time interval.
- Domain groups allow uniform batch creation, in which uniform number of URLs from each domain is present. This creates uniform fetcher queues which do not result in hung fetcher threads.

3.4 Resource limits

3.4.1 OS limits

Linux platform, imposes standards to be followed for running any software program which can be verified by running 'ulimit –a' in the Linux console. The maximum number of processes in a Linux environment is about 30,000. It also depends on the higher number of cores available for any the system. If fewer cores are available large number of spawned processes has to wait for CPU time.

The proposed crawler is implemented on node js which has 1GB space size limit on a 64-bit system. However, these limits can be increased but is advisable to scale the crawler more horizontally than vertically. Nodejs supports non-blocking I/O. But in case of unregulated I/O operations, the usage of event emitters shoots above the existing limit, resulting in memory leaks. In crawler the entire work load is dependent on child and parent process communication i.e. child manager and the fetcher processes. But, in Linux 8kb limit is present at a single pipe communication. Subsequently buffering is done to overcome this limit but if very large job files are used then it would take lot of time to send job files to fetcher process.

3.4.1 Database limits

The requirement for running a distributed crawler expects a distributed database setup with appropriate shards and replica configuration. So to achieve the same in the proposed crawler, MongoDB is used in distributed environment. However, the issues dealt with database limits are mentioned through the following:-

- If number of connection in a pool are increased more than 10,000 or so. The amount of CPU activity increases very much due to huge number of spawned connections to the database [12].
- File limits on OS creates problem with read/write speed, if database size is huge [12].
- Storing file size exceeding 16MB is not possible in MongoDB. Hence MongoDB's GridFS is used to store large files by dividing them into chunks. To get good performance it is advisable to use GridFS on a sharded environment [13].
- Inserting individual URLs in MongoDB collection results in huge number of open connections. To avoid this, bulk insert was used. So while inserting 1000 records without the bulk API, 30 operations per second was achieved. Whereas with the Bulk API an estimated 4000 operations per second is observed [14].
- Key size in MongoDB document adds to the size of the document. Hence using short key name is advisable. By creating a mapping between the real key and the key in the crawler reduces the size of database by 30%.

3.4.1 Network limits

Distributed crawling in Linux requires a better network layer for effective politeness in crawling. It is also observed that many crawler services are banned due to not following the rules in robot.txt. A technique used to resolve this is user agent spoofing. However it is not a permanent solution. So, to provide a discrete solution, a rotating proxy server is used. These comprise of a pool of IPs which are used to distribute the requests from the local network. It is found that in case of the sockets created, a total of 470 sockets can be utilized per second. To increase the number of sockets per second, adjustment can be made in the given default value:-

net.ipv4.ip_local_port_range = 32768 - 61000
net.ipv4.tcp_fin_timeout = 60

Thus, (61000 - 32768) / 60 = 470 sockets per second

4. EXPERIMENTAL OBSERVATIONS

In order to draw a comparison between Apache Nutch and the proposed crawler, few experiments were performed. This demanded for a web server which would generate random webpages with links. Simulating the following parameters: varied status codes, RSS links in meta, author links, timeouts in requests, alternate language links, various content types such as PDF, etc.

Hence following test condition was set on Apache Nutch and the proposed web crawler:-

- Fetcher threads: 10
- Politeness interval: 5 seconds.
- Size of seed file: 10,000 URLs
- Batch Size: 5000
- Concurrent Connections on each domain: 100
- Test Duration 5h

The URLs in the seed file given to crawler were added to /etc/hosts file and pointed to the established web server.

Table 1. Web crawling performance statistics of Apache Nutch 2.1 with HBase Backend, over 5h of web crawling

Parameter	Value
TOTAL URLs:	835581
Unfetched URLs	778035
Fetched URLs	57546
Duplicate URLs	1692
URL batches used	10
URL batches generated	10

Table 2. Apache Nutch crawling cycle performance over 10 iterations of web crawling which took ~5h

Step	Average Time Taken	Average Memory Consumption	Average Virtual Memory Consumption	Average CPU usage
Inject	27 min	126 MB	1.6 GB	6%
Generate	32 min	495 MB	2.4 GB	13%
Fetch	38 min	1218 MB	4 GB	42 %
Parse	13 min	700 MB	2.2 GB	11%
CrawlDB -update	18 min	1523 MB	2.6 GB	16 %
Average per iteration	128 min	812 MB	2.52 GB	17%

Average Fetch interval recorded: 12 seconds.

Time difference of 12 - 5 = 7 seconds. This is resulted due to the time taken by nutch's sequential steps to execute the crawling process.

Nutch crawl cycle fig 6. is sequential and each is dependent for previous steps.

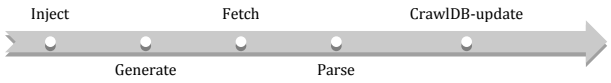

Fig 6: Nutch sequential crawl cycle

Thus, for a major fraction of time when fetcher is not used, the web domains receive no requests.

Table 2. Performance statistics of the proposed crawler over 5h of web crawling

Memory	1528 MB
Virtual Memory	1.3 GB
CPU	48%
TOTAL URLs:	1449456
Unfetched URLs	1296632
Fetched URLs	152824
Duplicate URLs	2260
URL batches used	32
URL batches generated	62

Average Fetch interval recorded: 7 seconds.

Hence the average fetch interval for the proposed crawler is close to the value set in configuration i.e. 5 seconds.

The above experimental results showcase the following:

- In the proposed web crawler the number of discovered links and the crawled pages are more than the Apache Nutch crawler running with the same configuration.
- The average fetch interval in the proposed crawler is close to the value set in the configuration which suggests that the fetch time is maximized.
- Using domain groups in the proposed crawler reduced the amount of information needed to hold in the main memory. This can be inferred with less virtual memory and memory usage in the proposed crawler.
- As, Apache Nutch progresses the time taken to create next URLs batch increases. But, in the proposed crawler the usage of in-memory Job File URL cache and the parallel Job File creator, was successful in creating more batches while running in the background.
- Apache Nutch generates batches and consumes them subsequently however; in addition to that the proposed web crawler generates batches for future jobs too. Which can be ran by adding more bots to the cluster.

5. INCONSISTENCY OF WEB CONTENT

The data arrangement on the web domains vaguely varied. It is generally far from the expected ideal. Hence requires regular human interference to avoid duplicity and monitor consistency of links. Few inconsistencies are listed below:-

- Not removing old links and leaving 404 as a response. This results in resource wastage.
- Not providing robots.txt and if they do, all the bots are banned.
- Sitemap files have old URLs which are redirected to new URLs.
- Mime type mismatch [15] is also prominent. It refers to having different content from the extension mentioned.
- Difference between the crawled content and the visible content, on a web browser due to dynamic content delivered by AJAX [15].
- In some discovered URLs only the order of the parameters are changed. For e.g.

  http:///www.exam.com?a=10&c=5
  http:///www.exam.com?c=5&a=10

  both are same.
- Some parameters are useless and results in new URLs but points to the same content. Not using redirects for the same content is also observed.
- Wrong HTTP content headers provide incorrect information regarding content-length, content type, date modified etc.[15]
- Malformed HTML markup.[15]

6. RESULTS

This paper addresses issues related to implementation of a distributed web crawler. It consists of component based architecture for a web crawler, along with selection policies and politeness techniques. Many open source crawlers still face problem while crawling larger data links (in millions). Also various issues in deployment of the crawler, managing crawled data, re-crawling links for update purposes, etc. has also been noticed. So in-order to address these problems a holistic component model is discussed which includes child manager, bot manager and the cluster manager. These include various sub parts which comprises of parsers, loggers, and processes which helps in generating a distributed breadth first crawler to counter the shortcoming in existing crawlers. Resource utilization still happens to be one of the challenges faced in web crawling. While working on web crawler certain issues related to network, OS and database were realized. To

overcome this few techniques as mentioned alongside each resource limit are implemented.

Thereby, describing a robust architecture and implementation details for a distributed web crawler with some preliminary experiments and results.

7. FUTURE WORK AND CONCLUSION

Addition to the work accomplished here, following are the areas that can be explored for future prospects. In recent time there has been a lot of discussion on Deep Web Crawling and AJAX crawling. These hold active interest and opens scope for more detailed and accurate web crawling. This inclines towards the field of Artificial Intelligence, empowering the spiders with human like selection intelligence.

Another major open issue is a detailed study of the scalability of the system and the behaviour of its components. This could probably be best done by setting up a simulation testbed. The main interest in using the web crawler is to look for prominent challenges in web search technology.

8. REFERENCES

[1] Shkapenyuk, V. and Suel, T. (2002). Design and implementation of a high performance distributed web crawler. In Proceedings of the 18th International Conference on Data Engineering (ICDE), pages 357-368, San Jose, California. IEEE CS Press.

[2] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In 7th Int.World Wide Web Conference, May 1998.

[3] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In 10th Int. World Wide Web Conference, 2001

[4] Web Crawling, By Christopher Olston and Marc Najork Foundations and Trends R in Information Retrieval Vol. 4, No. 3 (2010) 175–246 c 2010 C. Olston and M. Najork DOI: 10.1561/1500000017.

[5] Common Crawl, "Common Crawl's Move to Nutch," http://commoncrawl.org/2014/02/common-crawl-move-to-nutch/

[6] Burton H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors.

[7] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pages 117–128, May 2000.

[8] .J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In Proc. of 26th Int. Conf. on Very Large Data Bases, pages 117–128, September 2000

[9] George Adam, Christos Bouras, Professor Vassilis Poulopoulos, Utilizing RSS feeds for crawling the Web Conference: Fourth International Conference on Internet and Web Applications and Services, ICIW 2009, 24-28 May 2009, Venice/Mestre, Italy.

[10] Chakrabarti, Soumen, Martin Van den Berg, and Byron Dom. "Focused crawling: a new approach to topic-specific Web resource discovery."*Computer Networks* 31.11 (1999): 1623-1640.

[11] Broder, A. and Mitzenmacher, M., 2004. Network applications of bloom filters: A survey. Internet mathematics, 1(4), pp.485-509.

[12] High Scalability, "10 Things You Should Know About Running MongoDB At Scale" http://highscalability.com/blog/2014/3/5/10-things-you-should-know-about-running-mongodb-at-scale.html

[13] MongoDB,"GridFS - MongoDB Manual 3.2" https://docs.mongodb.com/manual/core/gridfs/

[14] Compose, "Better Bulking for MongoDB 2.6 & Beyond –Compose an IBM company". https://www.compose.com/articles/better-bulking-for-mongodb-2-6-and-beyond/

[15] Castillo, Carlos, and Ricardo Baeza-Yates. *Practical Issues of Crawling Large Web Collections*. Technical report, 2005.