# ΣΛ: Rivaling Server Based Instances in Scale, Performance and Convenience

Utkarsh Jain (ujain6@wisc.edu)
Sai Rohit Battula (battula2@wisc.edu)
Jack Chen (jchen488@wisc.edu)
University of Wisconsin – Madison

## Abstract

Cloud computing has been a favorite for upcoming software companies to build their services upon for several years. With focus on increasing developer velocity, cloud providers allow them to host their applications without the need for existing infrastructure. In recent years, cloud providers have made the process even more convenient. A serverless model like AWS Lambda decouples the developer flow and server administration, along with excellent elasticity of computing resources. Lambda functions can scale very well but it does not mean they can be panaceas. Our prototype, Sigma Lambda, built on top of AWS Lambda shows that it delivers on the promises of high parallelism, low latency and cheap costs by AWS.

## 1. Introduction

A serverless model does not imply that there is no server running your code. It is a way for application developers to transfer the administration, risk and provisioning of computing resources to the cloud provider. With Lambda, you still write your code, but now it can be configured to run exactly like a function call. When your Lambda function is triggered by an event, it's run in stateless containers - ephemeral and fully managed by your cloud. Users can execute thousands of containerized functions and use them at millisecond granularity.

Amazon Lambda, Microsoft Azure Functions and open source serverless platforms like OpenWhisk majorly benefit due to their ease of elasticity. When shifting your application to the cloud, it can be hard to determine what the future load for an application might look like. Moreover, the application could face sparse traffic with quick bursts rather than a consistently balanced experience. In these cases, it becomes harder for a user and the cloud provider to decide the optimal amount of computing resources. Lambda offers a solution with its pay-per-execution model, enabling fine grain and on-demand control over your resources; thereby preventing both overutilization and underutilization.

AWS provides functionality to add trigger events through API endpoints and configurable events across its service stack. Since functions run at every triggered event, achieving a high degree of parallelization becomes convenient as the application scales. These two strengths make it appealing for applications and make it ideal to build a web crawler on top of it. The web is a dynamic space consisting of billions of pages. A web crawler works by repeatedly taking a URL and extracting all the links from the HTML page received in the HTTP response.

This process can be sped up by extracting and crawling multiple links at the same time. When Google came out with the initial version of its search engine crawler, each node could open over 600 open connections [4]. Mature open-source web crawlers like Mercator, Apache Nutch and IBM have been able to crawl millions of pages using 10-15 machine clusters [4]. This type of workload can benefit greatly from a distributed and scaled configuration. Our intent in this paper is to design a model to develop a highly parallel crawler architecture on top of serverless platforms. We run metrics that demonstrate how AWS Lambda can achieve same or even better throughput at a lower price and good ease of use in applications that demand scalability.

Section 2 describes the initial challenges of designing a web crawler, and what previous server-based implementations have done to create parallel web crawlers. Section 3 goes over our implementation of the prototype and how the

challenges along the way shaped our backend architecture.

Section 4 presents an evaluation of our implementation through benchmarks and explain how well it fares against other web crawlers. Cost is an important factor for business enterprises and these results can convince them to gain advantages with Lambda's pay-per-use model. We report on number of concurrent Lambda executions, crawl rate and queue performance to complete our benchmarks. Section 5 contains various insights about our results and further discussion.

In summary, the paper has the following contributions

- A prototype that leverages scalability and elasticity of serverless models.
- Determine the capability of serverless models to run highly parallel jobs.
- Compare running costs and other metrics to evaluate viability.

# 2. Background

## 2.1 Web Crawling

The major motivation towards designing better models of distributed computing for web crawlers stems from the ever-increasing size of the World Wide Web. Any web crawler, like the UbiCrawler, starts with a single or a set of seed URLs [5]. This process repeats recursively where multiple crawling processes retrieve links from a queue of URLs. Hence, a crawling process can easily grow into a computationally expensive process requiring a lot of hardware resources. Due to the enormity of the web, it becomes even more imperative to design a parallel crawler since a single process can't take this burden. Full-fledged applications and parallel web crawler applications incorporate optimizations like spatial awareness of the web or consistent hashing to distribute links across servers so as to prevent flooding them with TCP connections.

One of the newer designs that came out in the recent years is Apache Nutch. It promises impressive scalable results by relying on the Hadoop file system for batch processing. Since Hadoop works on a server-based model, the cluster can be deployed on AWS Elastic MapReduce (EMR) clusters without complicated configurations. Also, EMR cluster instances are priced by the hour and thus provide a good comparison to our prototype (see §4).

These parallel distributed models offer high degree of parallelism and use several things to get the best output. We now discuss how the notion of a Lambda provides high scalability and efficient resource allocation without the pain of provisioning your own servers.

## 2.2 Serverless computing - Amazon Lambda

In the serverless computing model, you run your code on your cloud provider's hardware. Software applications can be broken down into smaller microservices in the serverless model. Running your code literally means invoking your code like a function call. The Lambda functions act as API endpoints that can be activated through HTTP GET or PUT requests. These endpoints execute the correct handler responsible for executing the logic, i.e., the code written by you. In-app events like modifications to an Amazon S3 bucket or SQS queue (like in our implementation) can invoke Lambda functions too. The stack is split into three parts – the API trigger, function logic, and backend AWS resources (DynamoDB, VMs, messaging queues etc.). The CPU time share allocation to a Lambda function is proportional to the amount of memory allocated at configuration. This is evident in cases where allocating insufficient memory resulted in seven times slower performance than expected [1].

We would now discuss some nuances of Lambda that explain how serverless computing achieves such high scalability. Upon each invocation, a separate lambda function can spawn within milliseconds inside a VM pool in Amazon's cloud. Investigations by Wang *et al* reveal that Lambda is capable of automatic scaling and can spin up N concurrent running instances to match the number of N concurrent invocations [11]. This limit goes as high as 1000 (in most regions), which can be further increased to 5000 by requesting their support team. Although, this depends upon your actual usage and the available resources on Amazon's side. This fully managed elastic and automatic scaling is one of the main advertised benefits of serverless platforms.

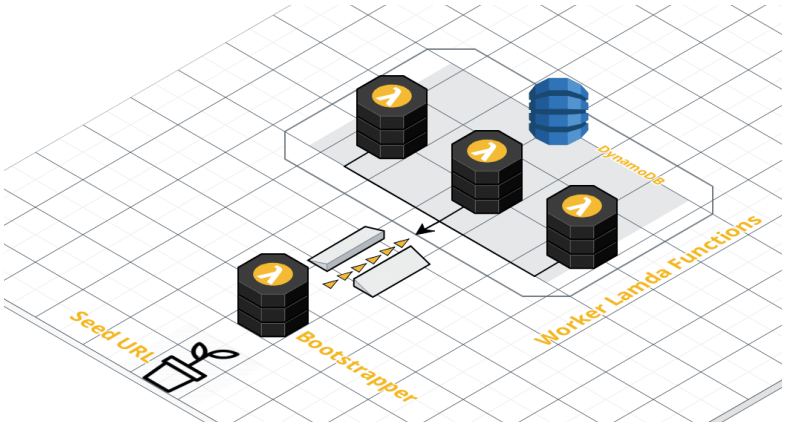


**Figure 1:** Backend architecture on AWS

A caveat of running functions in response to trigger events is that each invocation incurs startup and teardown overheads. The time it takes for Amazon to start the handler inside a container is referred to as the 'cold-start' latency. Amazon keeps a pool of ready VMs that keep the application warmed-up for future invocations. Coldstart events on a new VM is not much worse than a cold start on an existing VM. The median latency for both cases was 39ms [6].

## 3. Design

The architecture of our web crawler has been shown in Figure 1. This section describes how various steps of the crawling process are handled. The code contains functions written in Python and uploaded as a deployment package into a Lambda handler which executes the code. Each Lambda function is handled by an entity called the Lambda handler. Every Lambda handler has event and context parameters for input.

### 3.1 Lambda

The design can handle as many concurrent running instances as Amazon allows (500 - 3000) [9]. It is divided into two layers of lambdas as shown in Figure 1. A bootstrapper lambda is responsible for downloading the first set of URLs from the seed URL and placing them into a storage service. The storage service could be any middleware or a database which could share the state across Lambda functions. What storage service we made use of would be covered shortly but we refer to it as the working set for the sake of this discussion. We use a seed URL that is placed into the working set by the bootstrapper Lambda. We made a worker Lambda function which is responsible for taking in the URLs stored in the working set. The retrieved URLs will be requested (HTTP GET) and the response is parsed for new links. These new links are again placed in the working set, for new invocations of the worker Lambda to handle them. One important thing to note is that duplicate links could be retrieved during the parse phase and could be placed in the working set. Hence, to remove the overhead of crawling duplicate links, we had to store the crawled links persistently. This persistently stored source should help us detect duplicates.

### 3.2 Working Set

Our aim while deciding the service for the working set is that it should be able to support requests issued by multiple worker Lambda instances running in parallel. So, the ability to support/trigger a lot of concurrent Lambda instances was important. This could be issued by configuring event sources to trigger Lambda functions. As we have noted earlier, the trigger could be issued using an endpoint or configuring an AWS service with support for Lambda triggers. We needed additional state transfer to maintain endpoints, hence we stuck to using

another AWS service which would give us the ability to store state and trigger Lambda functions to augment the serverless nature of the architecture. We made use of AWS SQS queue to realize the goals mentioned above.

Amazon SQS is a fully managed queue service enabling easy communication between Amazon components. There is no limit on the number of messages, and you can achieve a high number of messages being read concurrently. The queue acts like a pipe carrying the URL flow with no FIFO ordering. Amazon recently added the functionality to trigger a Lambda function on a queue insert. If a URL could not be crawled because the Lambda failed to spawn or the TCP connection timed out, the SQS queue makes the URL visible again in the queue. This was really helpful to our prototype since that resulted in the automatic trigger of new Lambda functions. This had a corollary that thousands of parallel crawling processes were spinning without any complex synchronization between them. Furthermore, AWS took care of removing elements from the queue upon successful retrieval by a Lambda function. We didn't have to handle the removal explicitly.

When the seed URL is inserted into the queue by the bootstrapper Lambda, our web crawler function is executed for the first time. The worker Lambdas extract links from the queue, parse the HTML, and put them back in the SQS queue again. The SQS queue offers a number of configuration options such as message retention period, message visibility etc. When a Lambda trigger is setup for an event issued by the queue, AWS allows us to configure the batch size. This batch size for our case corresponds to the number of URLs the queue passes onto the event parameter of the worker Lambda function. Essentially, the batch size is the number of URL inserts it takes to trigger a new Lambda function.

### 3.3 Persistent Store

Our goals for choosing the persistent store for our already crawled links were:

- low read (for duplicate detection) and write latencies.
- Scalability to support inserts and reads from a vast number of Lambdas running in parallel.

We made use of DynamoDB given its virtually unlimited throughput, low latency and high scalability [12]. It further, lets users enable auto scalability for reads and writes making it more convenient.

We had the option to use DynamoDB as the working set in the place of SQS because they both can trigger Lambdas on each insert and can support concurrent executions. The deciding factor was the number of concurrent executions possible with each option. Both storage options are consumed by AWS Lambda, that continuously polls them for newly inserted records.

The problem is, the unit of concurrency for each of these options. In a DynamoDB database, that unit is the shard. If your stream has 100 active shards, there will be at most 100 Lambda function invocations running concurrently. Hence, Lambda processes each shard's events in sequence. While, the SQS's unit of concurrency was a configurable batch size. Multiple sources showed this implementation and mentioned DynamoDB as an event source being a bottleneck in terms of the horizontal scaling expected out of Lambda functions [9]. Hence, we turned towards AWS SQS. For Lambda functions that process Amazon SQS queues, AWS Lambda will automatically scale the polling on the queue until the maximum concurrency level is reached. This on-demand polling nature gives the edge to SQS. AWS Lambda's automatic scaling behavior is designed to keep polling costs low when a queue is empty while simultaneously enabling you to achieve high throughput when the queue is being used heavily [9].

Lambda has this great feature where it dynamically scales function executions as incoming traffic increases. For the US West (Oregon), US East (N. Virginia) regions, up to 3000 concurrent executions are possible.

### 3.4 How it works?

AWS Lambda service does a long polling on AWS SQS, which signals back AWS Lambda service new URLs (number of URLs is configurable, from 1 to 10) when they are ready. AWS Lambda then invokes a new worker Lambda function to process those new URLs.

When the worker Lambda is invoked, it first tries to insert the URLs into our database. By making the URL as the primary key, we prevent duplication. After checking for deduplication, the retrieved HTML page from the HTTPS response is parsed for <a> hyperlink tags and added to the queue. Each worker Lambda dies after going through the dispensed links and placing all the URLs found on their respective HTML pages to the queue. This means that we only pay for the amount of computing resources used.

Our code has been written in Python, using the Boto3 library to access AWS resources. Boto is the Amazon Web Services (AWS) SDK for Python, which allows writing software for accessing Amazon services such as EC2 and Lambda. DynamoDB is included in the AWS bundle supported by Boto. Boto provides an easy to use, object-oriented API as well as low-level direct service access for all AWS services mentioned above.

# 4. Evaluation

We evaluate Sigma Lambda's crawling rate and costs with other open-source web crawlers. Apache Nutch is a mature and tested web crawler that relies on Hadoop, which is great for batch processing loads as seen in crawling.

We compare metrics generated by AWS CloudWatch based on cost and the number of URLs crawled in a certain period of time. Our experiments are deployed on AWS and both are started with a single seed url: "https://www.youtube.com". Each Lambda function is configured with default 128MB memory for the code and additional dependencies. Each newly invoked function receives 10 links at a time from the SQS queue. Timeout for each instance is 59 seconds, at which point AWS Lambda terminates execution of the function. We chose our service region in Ohio, which imposed some limitations to the service we received from AWS.

We run Apache Nutch 1.15 on a cluster provided by AWS Elastic MapReduce service (EMR). The cluster consists of nine m4.large (2.3 GHz Intel Xeon, 8GB RAM) instances (one name & eight data nodes) with Hadoop 2.8.5. The main reason why we chose AWS EMR is that it allows an easy way to increase slave nodes. This is comparable to the level of easiness to scale AWS Lambda. We also attempted configuring our own Hadoop cluster with cheaper EC2 instance, t2.small, which has more complexities and will be explained in later section. We set up both our crawlers to request and download all HTML pages they receive in a HTTPS response.

In conclusion, our prototype was able to crawl 335,899 links in 1 hour at a cost of $1.88. While, Nutch was able to crawl 10,348 links in 2 hours at a cost of $2.34.

## 4.1 Scalability evaluation

Maximum number of concurrent executions is an important parameter to evaluate scalability. This number represents the maximum number of web pages a crawler can access at the same time, conveying the maximum workload a crawler can shoulder. In our implementation, as more and more worker Lambdas insert links into the queue, the SQS responds proportionally by invoking new functions up until it saturates its concurrency limit. As shown in Figure 3, the number of concurrent executions saturates its limit. Moreover, there is a plateau around ~430 concurrent executions. The number of concurrent executions has a trend to increase throughout the entire crawling process. Since most web pages contain more than 1 link, we expected an exponential increase in number of concurrent executions. This is only somewhat true in the graph between 20:45 to 20:55. When the number was about to reach default initial at 20:55, the increasing rate was throttled down. The automatic scaling recognizes traffic workload but responds slowly to the increase of traffic.

By calculating every 5-minutes average number of concurrent executions from 20:30 to 21:15 in Figure 3, we can obtain the average number of concurrent executions in that hour, which was 336, about 67% to the default maximum.

Nutch can scale easily if deployed on AWS EMR. It scales whenever the underlying Hadoop cluster scales. By configuration, each slave node runs 50 fetcher threads. Achieving the maximum of 400 concurrent execution requires 8 slave nodes. AWS EMR allows a maximum of 19 slave nodes per account. The number of concurrent executions can be configured easily

by submitting different requested number to AWS console. However, shrinking the size of the cluster could incur data transfer from deleted nodes to existing nodes, which could increase overall costs and complexities. Increasing number of slave nodes has little effect on the crawling experiment.

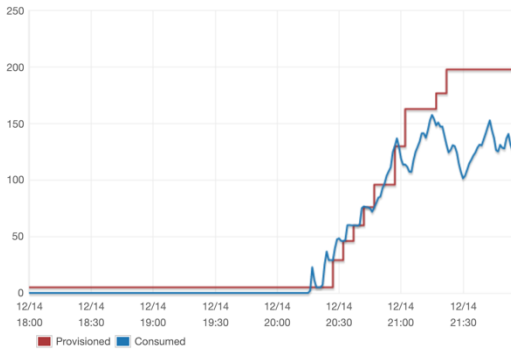


**Figure 2:** Number of URLs/second

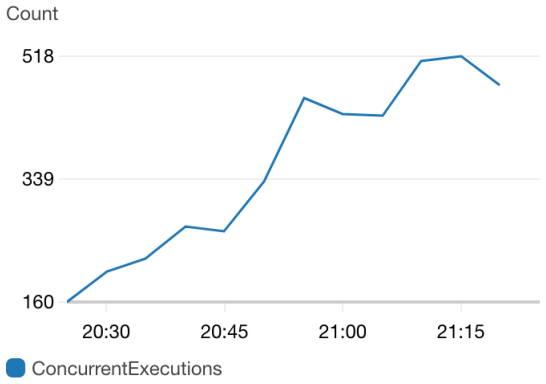


**Figure 3:** Concurrency of Lambda functions over time

## 4.2 Performance

AWS Lambda can spawn worker functions very quickly, and the rate of instantiating new functions is very high. In Figure 4, the graph measures the crawl rate as number of parallelly executing functions increase. Since the number of links inserted into the queue grows with a sharp linear spike, so does the number of invocations. The snowball effect of this can be seen in the gap between the orange and blue curve in Figure 4.

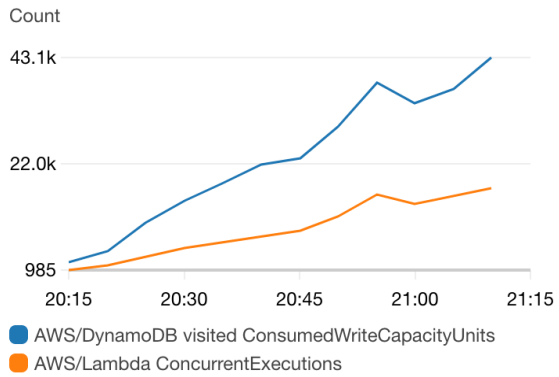


**Figure 4:** Number of concurrent Lambda functions over time

Consumed write capacity units measures the throughput used and indicates the rate at which URLs are being written to the persistent state. Figure 2 also graphs the number of writes to the DynamoDB in writes/second. Each Lambda function crawls almost 200 links each, so you can see how increased number of Lambda functions spinning per second results in higher number of URLs being written to the persistent store.

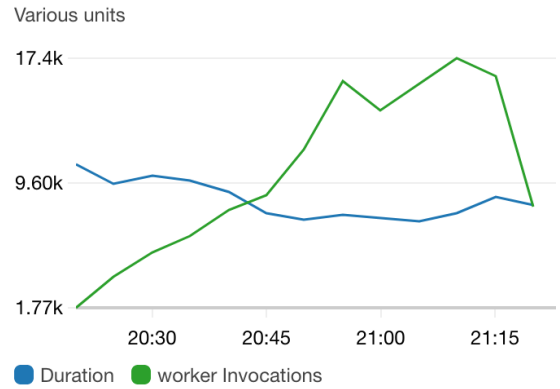


**Figure 5:** Avg. duration of functions (ms) vs. number of Lambda invocations over time

Figure 2 also suggests a sudden linear increase in the number of units/sec (URLs/sec in our case). This is the result of the sudden increase in concurrent Lambda invocations in the start. The red line indicates how auto scaling helps

provisioned write units of DynamoDB catch up to the writes issued by Lambda functions.

There are no bottlenecks in this design as SQS can trigger Lambda functions as fast as it can receive them. Moreover, DynamoDB provides high concurrent reads for checking URL duplication before crawling. The maximum sum of concurrent executions in a 15-minute period was close to 11,000. In a one-hour period, we made over 80,000 requests to our Lambda functions and crawled nearly 335,899 unique URLs. The throughput was about ~150 URLs per second as Figure 2 indicates. We were able to only reach 518 concurrent Lambda functions, and the throughput corresponds to this. Of these 80,000 invocations, a large number of spawned instances paid the overhead of fetching already crawled links.

Doing 80,000 5-minute jobs in parallel takes roughly about an hour and observed startup and teardown overheads have been low. To mitigate cold start latencies, Amazon caches container status and file system information. The warming up of the workload can be seen in Figure 6.

The high crawl rate achieved is attributed to the high rate of inserts into our SQS queue apart from automatic scaling. Figure 6 shows the number of inflight messages in the SQS queue. With more than 735k URLs in the queue, SQS is a strong addition to the Lambda architecture.


**Figure 6:** Number of URLs in SQS over time

Apache-Nutch divides crawling process into several Hadoop jobs in a sequence. We set maximum fetch list size to 50,000, so that Nutch can crawl as many as it can in each crawling round; timeout for fetching job is set to 300 seconds, and number of threads for each slave node is set to 50. The most time-consuming job is fetching. In order to compare against our prototype, Nutch started crawling with the same seed URL of YouTube's homepage. Time spent on fetching greatly exceeded the time consumed by other jobs as shown in Figure 5. Fetching consumed 44.2% of total execution time, which is 483.2 seconds. In two hours of crawling process, Nutch fetched 10,348 links. Since Nutch only spent 44.2% in fetching, the throughput is 2.54 URLs per second.

### 4.3 Economic factors/pricing

AWS Lambda is priced based on the amount of memory allocated to the function. We chose 128 MB, and each invocation of Lambda function costs $0.007488 for the compute seconds (the granularity of charging is 100 milliseconds). We fetched 335,899 distinct URLs for a one-hour workload. The total cost for the crawling process was $1.88, as reported by the Amazon Console.

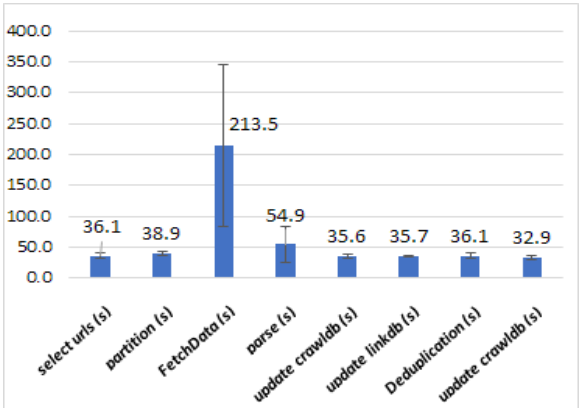


**Figure 7:** average time taken by crawling jobs without indexing (seconds)

On the other hand, for traditional AWS compute services, prices can vary from $0.03 to $0.3 per machine per hour when the cluster is standing by. When machines are running, the on-demand pricing ranges from $0.1 to $5.0 per machine per hour. In our settings, we employed the cheapest possible machines. Crawling 10,348 distinct URLs took 2 hours with a of cost $2.34. The breakdown was 54 cents for the EMR cluster and $1.80 for on-demand computing.

We thus compare the two results by normalizing the data for Nutch. Nutch can only fetch 11705 distinct URLs per hour on average

(excluding the time spent on other jobs), and cost $1.176 per hour. In our experiment, Lambda can perform crawling at 37% higher price while fetching ~28.7 times more URLs than Nutch.

### 4.4 Convenience of usage

These new serverless platforms like Amazon Lambda, Microsoft Azure Functions and Google Cloud functions strive to make their tools easy to use and fast to deploy. Amazon Lambda can be integrated with a large number of Amazon services like streams, databases, and other cloud events. This makes the platform very easy to use as compared to deploying Nutch on an EC2 instance. There are multiple abstractions that let you build complex structures easily.

Native Apache-Nutch requires little configuration and can start crawling with pre-installed crawl script. It will detect whether Nutch is being run in local mode or distributed mode. Other useful tools to view crawler status and crawl database statistics are also pre-installed.

## 5. Discussion

Before setting the fetch timeout to 5 minutes, Nutch may spend more than 2.5 hours in fetching. The reason for the delay is that some slave nodes are making requests to slow domains, which keeps the entire cluster waiting until either the fetching job times out or the slave nodes finish fetching. The long delay can generate additional costs while no URLs are being fetched. Those kinds of costs were rarely incurred in Sigma Lambda. Due to the difference in architecture, if one invoked function is blocked, other invoked functions can still maintain the chain of invocations. Moreover, we did observe that during each crawling round of Nutch, number of fetched pages may double or quadruple while finishing one crawling round usually takes 15 minutes in our configuration. Yet, the Lambda's number of invocations reached its plateau after 30 minutes of execution. Due to limited funding, we are unable to continue running the experiment in a long-term manner and explore how two crawlers behave in a long crawling task. Based on the results from another paper, Nutch ran on a cluster of 16 machines. Configured with a slightly optimized scoring filter, a smaller fetch list, and more diverse seed URLs, it was able to get quarter billion URLs with costs under $600 in a 40-hour period [8]. To accomplish such a grand target, Sigma Lambda would spend more than $2,200 dollars based on our current statistics.

Installing Apache-Nutch on Hadoop cluster deployed on cheap EC2 instances is possible. However, our attempts with t2.small instances, which cost only $0.017 per hour, were not successful. To make a cluster working properly requires experience on both configuring Hadoop and working with AWS services. One wrongly configured name or data nodes can cause network flush between data and name, which can significantly increase bills.

It is worth to note that Lambda gives 1M requests or 400,000 GB seconds of free usage every month. After that, every 1M requests costs 20 cents [1]. This is a nice feature that we did not include in our evaluation because this is a promotion strategy of the service provider, which might not persist in the future.

In the implementation of Sigma Lambda, we ignored the failure of fetching (response with status code 404) and randomly generated URLs that represent the same web page. While Nutch handles the former situation, how it handles the latter is not clear to us. From Nutch's statistics, fetching 10,348 distinct URLs could encounter 4,074 gone pages (404 pages) and 1,211 duplicated pages. Even if those pages are counted, and Nutch fetched 17764 pages (divided by 2 x 44.2% to exclude time spent on other jobs) in one hour, its result was 5.3% of Sigma Lambda's in the short run.

Lastly, we are not certain whether the underlying system of Lambda and AWS EMR may have influences on our results or not. We can only blindly trust AWS to provide similar computational and I/O performance on both Lambda and EMR services.

## 6. Related Work

Serverless computation has attracted more attention in recent years. It is proposed as a new model of Function-as-a-Service (FaaS). For FaaS customers, serverless computation tools allow them to focus on the cost of their module design without too much concern on system

environment and system scalability [2]. Some interesting applications have been made trying to leverage these nice properties of serverless computation. For example, fast interactive video encoding and JPEG recompression in a distributed system could be accomplished potentially with reasonable charges [3]. Tasks that benefit from parallelism can potentially be implemented or facilitated by serverless computational platforms such as AWS Lambda, Google Cloud Functions, and Azure Serverless Computing, while keeping their customers satisfied with the price points.

On the other side, distributed web crawlers have some potential to utilize some nice properties offered by Lambda. Our first inspiration is Google, who built distributed web crawlers in order to "scale to hundreds of millions of web pages" [4]. It achieved a downloading rate of over 100 web pages per second [4]. However, the paper only discussed web crawlers in brief. More papers were published later on to summarize the design of web crawlers. There are generally two approaches based on the mechanisms to partition URLs: dynamic assignment and static assignment [6].

Apache Nutch is an open-source distributed web crawler that follows static assignment pattern, where fetching only happens after URLs are assigned to different worker machines. It uses MapReduce as underlying architecture to increase scalability. *Nioche*, in his experiments, ran Nutch on a very large cluster of 400 EC2 instances [7]. The workload ran for SimilarPages.com and was able to download and parse 3 billion pages. Sigma Lambda follows the dynamic assignment pattern, where fetching and assigning URLs can happen at the same time. It's economically not feasible to conduct a long-term experiment with more than thousands of concurrent executions for Lambda at this level for now. We can only estimate the performance and cost based on our experiment results.

## 7. Conclusion

Serverless computation proves its merits in providing highly scalable services at reasonably low expenses. While abstracting away concerns about operational environments, programmers can focus more on code and costs of their design. From the perspective of small applications, serverless computing is a very attractive solution for similar computer and data bound workloads. However, we had limited resources and funding to experiment with millions of URLs. Indexing downloaded web pages and providing searching services are also part of a web crawler's job. We did not provide analysis for those aspects. More future analysis is needed to confirm the applicability of large web crawlers based on serverless computing architecture. For our scope, Sigma Lambda performs well: it consistently achieves high crawl and invocation rate while keeping running costs low.

## Acknowledgements

We thank Prof. Michael Swift for his guidance in the design and evaluation, and for commissioning this project. We also thank our reviewers Nick Daly, Gautham Sunjay and Mitali Rawat for their valuable comments.

## References

[1] V. Holubiev. *My Accidental 3-5x Speed Increase of AWS Lambda Functions*. December 11, 2016. https://serverless.zone/my-accidental-3-5x- speed-increase-of-aws-lambda-functions-6d95351197f3

[2] Baldini, Ioana, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, & P. Suter. *Serverless Computing: Current Trends and Open Problems*. 2017.

[3] K. Winstein. *Tiny functions for codecs, compilation, and (maybe) soon everything*. April 20, 2018. https://platformlab. stanford.edu/Seminar%20Talks/2018/Keith%20Winstein.pdf

[4] S. Brin, L. Page. *The anatomy of a large-scale hypertextual Web search engine*. Computer Networks and ISDN Systems. 1998

[5] P. Boldi, B. Codenotti, M. Santini, & S. Vigna. *UbiCrawler: a scalable fully distributed Web crawler*. Software-Practice and Experience. 2004

[6] J. Cho, H. Garcia-Molina. *Parallel Crawlers*. *ACM*. 2002.

[7] J. Nioche. *Large Scale Crawling with Apache Nutch and Friends*. November, 2013. https://www.slideshare.net/digitalpebble/j-nioche-lucenerevoeu2013

[8] L. A. Lopez, R. Duen, S. Jodha Singh Khalsa. *Optimizing Apache Nutch For Domain Specific Crawling at Large Scale*. IEEE International Conference on Big Data. 2015.

[9] docs.aws.amazon.com/lambda/latest/dg/scaling.html#scaling-behavior

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht. Occupy the Cloud: Distributed Computing for the 99%. ACM. 2017.

[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift. *Peeking Behind the Curtains of Serverless Platforms*. USENIX. 2018

[12] Amazon DynamoDB - https://aws.amazon.com/dynamodb/