An Exploration of Serverless Architectures
for Information Retrieval

Matt Crane and Jimmy Lin

David R. Cheriton School of Computer Science
University of Waterloo, Ontario, Canada
{matt.crane,jimmylin}@uwaterloo.ca

ABSTRACT

Serverless architectures represent a new approach to designing applications in the cloud without having to explicitly provision or manage servers. The developer specifies functions with well-defined entry and exit points, and the cloud provider handles all other aspects of execution. In this paper, we explore a novel application of serverless architectures to information retrieval and describe a search engine built in this manner with Amazon Web Services: postings lists are stored in the DynamoDB NoSQL store and the postings traversal algorithm for query evaluation is implemented in the Lambda service. The result is a search engine that scales elastically with a pay-per-request model, in contrast to a server-based model that requires paying for running instances even if there are no requests. We empirically assess the performance and economics of our serverless architecture. While our implementation is currently too slow for interactive searching, analysis shows that the pay-per-request model is economically compelling, and future infrastructure improvements will increase the attractiveness of serverless designs over time.

1 INTRODUCTION

Servers, referring to both software stacks and the machines they run on, are central to the architecture of information retrieval systems. In the standard design, a search service waits for requests from a client based on some well-known protocol (e.g., HTTP or an RPC framework such as Thrift), executes the query, and returns the result. In a distributed search architecture, each server may only be responsible for a small partition of the entire document collection, and there may be many replicas of the same service, but servers remain the basic building block.

The advent of cloud computing means that physical machines are nowadays increasingly replaced by on-demand virtualized instances under a pay-as-you-go model. However, running a search engine still requires managing servers in some form. Even if there are no requests, one still needs to pay for some basic level of provisioning, in anticipation of incoming queries. As the query load increases, one then needs to provision more servers and load balance across them. Although there are tools to assist with scaling

up (and down) elastically, our goal is to explore alternative architectures that simplify management.

A new trend in cloud computing under the banner of serverless architecture or serverless computing aims to divorce the execution of stateless services from the server machines they run on (whether physical or virtualized). For example, Amazon's Lambda service lets a developer run code without provisioning or managing servers. The developer specifies a block of code that needs to be executed with well-defined entry and exit points, and Amazon handles the actual execution of the code—from a few times per day to thousands of requests per second.

This paper explores applications of serverless architectures for information retrieval and describes a search application built entirely using this approach with Amazon Web Services. Our key insight is that search breaks down into two components: postings lists that comprise the index and postings traversal algorithms that manipulate the postings to compute query results. The postings lists represent the "state" of the application, which we store in Amazon's DynamoDB NoSQL store. The "stateless" query evaluation algorithm is encapsulated in Lambda code that fetches postings of query terms stored in DynamoDB to compute query results.

The contribution of this work is the first application of serverless computing to information retrieval that we are aware of. We show that it is indeed possible to build a fully-functional search engine that does not require the explicit provisioning or management of servers. Experimental results show that our design yields end-to-end query latencies of around three seconds on a standard web test collection of approximately 25 million documents. While this latency is not acceptable for interactive retrieval today, the economics of the pay-per-request model is compelling. We believe that our design is interesting, and as serverless architectures gain popularity, infrastructure improvements will increase the attractiveness of our approach over time.

2 BACKGROUND

Serverless computing represents the logical extension of the "as a service" cloud computing trend that began in earnest a decade ago (even though precedents date back many decades to the advent of timesharing machines). Infrastructure as a service (IaaS) provides elastic, on-demand computing resources, usually in the form of virtual machines—Amazon's EC2 was the first and remains the most prominent example of this model, although Microsoft, Google, and many others have similar offerings. These cloud providers also offer storage and other infrastructure components (e.g., network virtualization) in a pay-as-you-go manner. Platform as a service (PaaS) raises the level of abstraction, where the cloud provider manages a complete computing platform—a typical example is Google App

Engine, which supports hosted web applications. Database as as service (DBaaS), such as Amazon's Relational Database Service (RDS), Microsoft's Azure SQL, and Google's Cloud SQL, provides managed database services that simplify provisioning, administering, and scaling relational databases in the cloud.

Database and storage as a service can be viewed as providing developers the ability to offload the management of "state" to a cloud provider. Many modern web applications centralize state in a database or some backend data store to simplify design and to support horizontal scalability. Therefore, most, if not all, application logic becomes stateless, in the sense that state is not preserved across multiple invocations of a particular functionality. Thus, the application just becomes a bunch of functions that access a common data store. If the responsibility of managing state is then pushed to a hosted cloud solution, then all that is left is a bunch of functions. In such an architecture, the developer does not really care how these functions are executed—hence, serverless.

Serverless computing does not actually mean that code can run without servers—but rather that from the developer's perspective, the execution of self-contained functions becomes someone else's problem, namely, that of the cloud provider. The developer does not need to worry about spinning up servers (or VM images), aggregating multiple execution instances to increase utilization, load balancing across multiple server instances, scaling up and down elastically, etc. The advent of lightweight containers with additional namespace virtualization and tooling, exemplified by Docker [11], makes serverless computing practical.

To date, most discussions of serverless computing take place in the context of redesigning user-facing applications in this paradigm. Such a decomposition is consonant with the "mircoservices" architecture that is in vogue today. For example, Hendrickson et al. [7] speculate about what it would take to rebuild Gmail in a completely serverless design, and the breakthroughs necessary to make it a reality. In this paper, we focus on the backend and explore what serverless information retrieval might look like.

3 SERVERLESS DESIGN

This section describes the design of our serverless search architecture, shown in Figure 1. We explain how index structures are mapped to DynamoDB and how the query evaluation algorithm is implemented using Lambda functions. At present, we have designed our system entirely around Amazon Web Services and thus vendor lock-in is a concern. Other cloud providers offer similar capabilities, although they are not as mature as Amazon's services. Cloud interoperability is an important issue in its own right, but beyond the scope of our work.

In this paper, we consider the JASS score-at-a-time query evaluation algorithm on impact-ordered indexes [9]. This approach has been shown to be both effective and efficient compared to state-of-the-art document-at-a-time approaches [4]. Query evaluation in JASS begins with lookup of postings corresponding to query terms. Each postings list comprises a sequence of decreasing impact scores, each of which is associated with a run of sorted docids (which we call a segment). To simplify our implementation, we currently do not compress docids [10]. Segments from postings lists of all query terms are sorted in decreasing impact score and processed in that
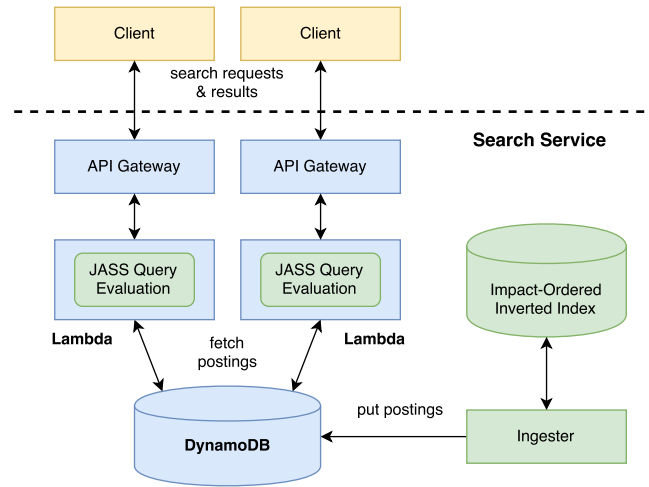
Figure 1: Our serverless search architecture. AWS infrastructure is shown in blue (Lambda and DynamoDB) and our custom components are shown in green.

order. For each segment, its impact score is loaded, and for each docid in the segment, the impact score is added to its accumulator. In JASS, the accumulators are implemented as an array of 16-bit integers, one per document, indexed by the docid. To avoid sorting the accumulators once all postings segments have been processed, a heap of the top k can be maintained during processing. That is, after adding the current impact score to the accumulator, we check if the document score is greater than the smallest score in the heap; if so, the pointer to the accumulator is added to the heap. After all postings segments have been processed, the top k elements are extracted from the heap and returned as results.

3.1 DynamoDB Index Storage

DynamoDB [5] is Amazon's fully-managed NoSQL store that supports a basic key–value model. One of its key features is that the user pays only for data storage and read/write operations. This pricing model is truly pay-per-request, in contrast to Amazon's Relational Database Service (RDS), which requires payment for server instances, regardless of query load.

DynamoDB has three core components: tables, items, and attributes. Tables store collections of related data. An item is an individual record within a table, and an attribute is a property of an item. In DynamoDB, items in the same table can have attributes that are not shared across all items. DynamoDB supports two types of primary keys: One attribute is selected as the partition key and is used internally by the service itself for data placement. Optionally, a second attribute can be selected as the sort key. No two items within a table can share a primary key, but DynamoDB supports additional indexes.

At construction time, each DynamoDB table needs to have a name and an associated primary key defined. Otherwise, the tables are schemaless, which means that neither the attributes, nor their types, have to be defined prior to data insertion. DynamoDB items have a size limit of 400KB, which is an important limitation we need to overcome (details below).

A naïve mapping from an inverted index to a NoSQL store would be to use the term as the partition key, and to store the postings for that term as the value. The issue with this design is that even for small collections, the size of the postings lists will exceed the 400KB size limit of DynamoDB items. Fortunately, the organization of impact-ordered indexes presents a natural way of breaking up the postings—by their impact scores. However, with sufficiently large collections, a postings segment (particularly for small impact scores) can still exceed the 400KB limit. To accommodate this we introduce the notion of "groups", an ordering of different runs of docids that share the same impact score. In DynamoDB, we use a hybrid sort key comprised of the impact score and the group number within that impact score.

Recall that for JASS score-at-a-time traversal we must retrieve postings for a term and a given impact score. Unfortunately, our hybrid sort key design does not make this easy to do. As a workaround, we created a secondary index on the postings table with the term as the hash key and the impact score as the sort key to support querying directly by impact score. Because there is no uniqueness constraint for primary keys in a secondary index, this approach works regardless of whether or not the postings for an impact score are split across DynamoDB items (i.e., different groups). In addition to the postings table, we created a separate metadata table, which stores the number of documents in the collection (necessary for the initialization of query evaluation) as well as a list of impact values that have postings for each term. This design allows us to avoid fetching non-existent impact scores.

Finally, we built an ingester program that takes impact-ordered indexes from an external source and inserts the postings into DynamoDB according to our design. Our current implementation is rather naïve and does not manage "hotspots" in the underlying DynamoDB table that develop when inserting many items with the same partition key, and hence does not achieve high throughput.

3.2 Lambda Query Evaluation

Amazon's Lambda lets developers run code without provisioning or managing servers, although creating a Lambda requires specifying the amount of memory that is available to each code invocation (up to a maximum of 1.5GB) and a timeout period (not exceeding 300 seconds). Code invocations are charged according to the duration of the execution, rounded up to the nearest 100ms in a very fine-grained manner. While there are no specifications of computational resources provided to execute the Lambda, both the network bandwidth [6] and the amount of processing power [2] have been observed to scale linearly with the memory requested.

Lambda code must be written in a supported language: JavaScript, Python, Java, or C#. However, there is no restriction against invoking code written in other languages. It is trivial, and indeed common usage, to bundle resources such as native binaries and libraries along with the function code itself. Our Lambda function is implemented in Python, which then invokes a program written in C++ that performs the actual query evaluation.

When an invocation request arrives at the API Gateway (a trigger that invokes a Lambda on HTTP events), Amazon is responsible for provisioning the necessary resources to execute the Lambda and managing its lifecycle. All of this happens without our intervention.
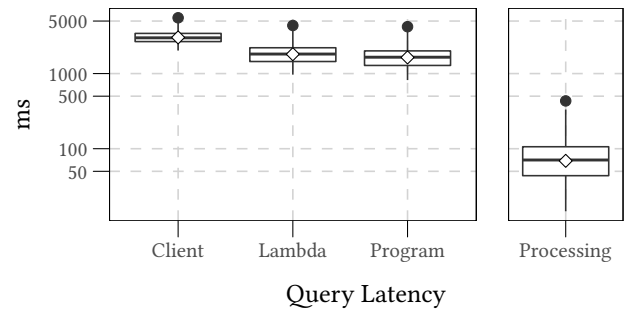

Figure 2: Performance of our serverless architecture.

Within the Lambda itself, our code first requests information about the number of documents and the impact scores for the query terms from the metadata table. After fetching this information, the accumulators and the heap are initialized, followed by the actual processing of the impact segments of the query terms in descending order. For each impact score, the DynamoDB requests are issued asynchronously, and the results are processed when available. While it would be possible to perform all requests asynchronously, this was not done since it would not yield a correct score-at-a-time traversal order. After processing has completed, the top k results are returned ($k = 1000$ in our experiments). Our implementation currently returns internal numeric docids instead of external (string) docids that are collection-specific.

4 EXPERIMENTS

To validate our design, we implemented the serverless retrieval architecture described in the previous section on the Gov2 collection, comprised of around 25 million web pages. For evaluation, we used topics 701–850 (with stopwords removed) used in the Terabyte Tracks from TREC 2004 to 2006 [3]. For expediency, we only ingested into DynamoDB the postings lists of the query terms.

4.1 Performance Analysis

We report experimental results in Figure 2, showing standard box-and-whiskers plots for query latency, with the mean shown as a white diamond. Latency figures are broken down as follows: "Client" is measured from the search client using the Unix command `time` (mean: 3087ms), "Lambda" is the billable duration as measured by Amazon (mean: 1887ms), "Program" is the internal timing by our query evaluation algorithm (mean: 1722ms), and "Processing" captures the amount of time spent performing query evaluation outside of waiting for DynamoDB requests (mean: 87ms). The difference between the "Program" and "Lambda" measurements captures the overhead of the Python Lambda invoking the native C++ binaries for query evaluation. The difference between "Lambda" and "Client" represents the additional overhead of invoking the Lambda itself and retrieving the results. Overall, everything other than the "Processing" measurement reflects overheads of the serverless architecture in various forms.

Even with all the "obvious" optimizations that we have implemented, end-to-end client query latency is longer than is typically considered usable for an interactive search application. To better contextualize these results, a recent open-source reproducibility

challenge organized by Lin et al. [8] reported a query latency of JASS under similar experimental conditions as 51ms (same collection, same queries, on an EC2 instance). This compares favorably with our "Processing" time, and the performance gap can be likely attributed to CPU differences in the underlying instances.

Overall, our experiments identified many sources of latencies in the current design, the biggest of which involves fetching postings from DynamoDB. There is substantial room for improvement, and we would expect that as serverless designs become more popular, Amazon would address these bottlenecks over time. Beyond DynamoDB latencies, there are a few obvious inefficiencies: for example, the invocation overhead of the C++ program can be eliminated if AWS supported C++ Lambdas. Furthermore, there is time wasted in needless data conversion—all Lambda requests and responses must be in JSON format and `binary` attributes in DynamoDB are encoded in base64, which is slow to decode. It would not be very difficult for Amazon to provide the developer more fine-grained control over serverless execution in these regards.

Beyond these experiments, there are several additional questions regarding our setup. In performance evaluations, it is customary to distinguish "cold" runs and "warm" runs, where the latter benefit from caching effects. Since both DynamoDB and Lambda are fully-managed services, this is difficult for us to accomplish as many aspects of execution are not as transparent as we would like. However, since our work is primarily a feasibility study, we defer these more detailed explorations to future work.

4.2 Cost Analysis

A key feature of our serverless design is the pay-per-request model and the automatic horizontal scalability of Lambda and DynamoDB in response to demand. In this section, we provide a cost analysis comparison of serverless and server-based architectures.

For a fair comparison, we once again turn to results from the reproducibility study of Lin et al. [8], which also examined JASS on the same collection and queries. On an EC2 r3.4xlarge instance, Lin et al. reported a query latency of 51ms on a single thread. Since the instance has 16 vCPUs, if we assume linear scaling, we arrive at a throughput of around 313 queries per second on a fully-loaded server. This instance costs USD$1.33 per hour regardless of load, which means that the cost is the same whether the server executes zero, one, or one million queries in any given hour. On the other hand, Lambda is charged on a per-request basis in increments of 100ms. The average billable time for our system was 1887ms per query, which translates to USD$0.000047951. DynamoDB storage is charged at USD$0.25 per GB per month plus additional costs for read and write operations. However, our usage levels remain in the DynamoDB "free tier" for these experiments, although a heavier query load would not substantially affect our analysis.

In Figure 3, we model the per-query cost in cents for the server-based and serverless architectures assuming the configurations above, as a function of query load in queries per second (qps). The Lambda design has a constant cost per query, while the EC2 instance becomes more cost-effective at higher loads, with the breakeven point around 7.7 queries per second. In addition, with Lambda we achieve (potentially unlimited) scalability without manual intervention. While a load of 7.7 qps seems low, consider that
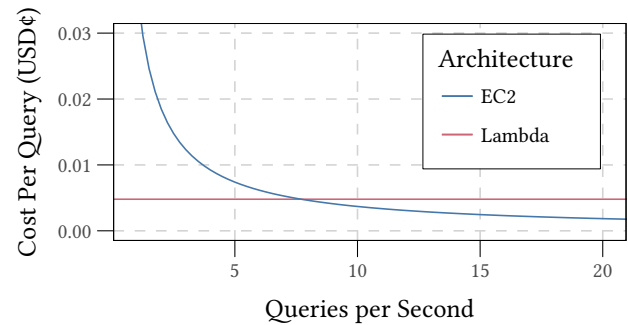

Figure 3: Cost of serverless vs. server-based architectures.

in the overview of the TREC 2016 Open Search Track [1], it was revealed that CiteSeerX receives nearly 100,000 queries per day, which translates into 1.2 qps on average. We venture that in all but the most demanding applications (e.g., commercial search engines), a serverless design would be compelling from a cost perspective.

5 CONCLUSIONS

Trends point to an inevitable move of computing to the cloud, and serverless architectures reflect this evolution. This work represents, to our knowledge, the first design of a serverless architecture for information retrieval. We readily concede that this initial iteration suffers from performance issues, although our cost analysis justifies the pay-per-request model for most search needs. We expect that future improvements in cloud infrastructure, along with additional optimizations in our design, will render serverless information retrieval increasingly attractive.

REFERENCES

[1] Krisztian Balog, Anne Schuth, Peter Dekker, Narges Tavakolpoursaleh, Philipp Schaer, and Po-Yu Chuang. 2016. Overview of the TREC 2016 Open Search Track.
[2] cecilia@aws. 2014. Re: Lambda CPU relative to which instance type? (9 Dec. 2014). Retrieved February 2, 2017 from https://forums.aws.amazon.com/message.jspa?messageID=588722
[3] Charles Clarke, Nick Craswell, and Ian Soboroff. 2004. Overview of the TREC 2004 Terabyte Track. In *TREC*.
[4] Matt Crane, J. Shane Culpepper, Jimmy Lin, Joel Mackenzie, and Andrew Trotman. 2017. A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation. In *WSDM*. 201–210.
[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*. 205–220.
[6] Vineet Gopal. 2015. Powering CRISPR With AWS Lambda. (25 Sept. 2015). Retrieved February 6, 2017 from http://benchling.engineering/crispr-aws-lambda/
[7] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *HotCloud*.
[8] Jimmy Lin, Matt Crane, Andrew Trotman, Jaime Callan, Ishan Chattopadhyaya, John Foley, Grant Ingersoll, Craig Macdonald, and Sebastiano Vigna. 2016. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *ECIR*. 408–420.
[9] Jimmy Lin and Andrew Trotman. 2015. Anytime Ranking for Impact-Ordered Indexes. In *ICTIR*. 301–304.
[10] Jimmy Lin and Andrew Trotman. 2017. The Role of Index Compression in Score-at-a-Time Query Evaluation. *Information Retrieval* 20, 3 (2017), 199–220.
[11] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014, 239 (2014), Article No. 2.