Serverless Data Analytics with Flint

by

Youngbin Kim

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Serverless architectures organized around loosely-coupled function invocations represent an emerging design for many applications. Recent work mostly focuses on user-facing products and event-driven processing pipelines. In this thesis, we explore a completely different part of the application space and examine the feasibility of analytical processing on big data using a serverless architecture. We present Flint, a prototype Spark execution engine that takes advantage of AWS Lambda to provide a pure pay-as-you-go cost model. With Flint, a developer uses PySpark exactly as before, but without needing a Spark cluster and only paying for the execution of individual Spark programs. We describe the design, implementation, and performance of Flint, along with the challenges associated with serverless analytics.

Acknowledgements

I would like to first thank my advisor, Professor Jimmy Lin, for his continuing guidance, remarks and engagement through the learning process of this master thesis. Furthermore, I would like to thank the readers of my thesis, Professor Bernard Wong and Professor Ken Salem, for reviewing my work. In addition, I would like to thank Michael, Jaemyung, Wei, Royal, Dr. Khuzaima Daudjee, and the rest of the Data Systems Group for their support. Finally, I would like to thank my parents and Hyeonjoo for their unwavering encouragement and support through all these years of school.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Serverless Architecture 4

1.2 Serverless Analytics 5

2 Related Work 7

2.1 Iris 7

2.2 Amazon Athena & Amazon Redshift Spectrum 8

2.3 Databricks Serverless 8

2.4 PyWren 10

2.5 Qubole Spark on Lambda 12

2.6 Summary 13

3 Architecture 15

3.1 Design 15

3.2 Flint Executor 17

3.3 Remote Storage 18

3.4 Overcoming Lambda Limitations 21

4 Experimental Evaluation **25**

4.1 Flint Executor Performance 25

4.2 Latency/Cost Tradeoffs 29

4.3 Discussion 31

5 Conclusion **33**

5.1 Future Work 33

5.2 Summary 34

References **35**

APPENDICES **37**

A PDF Plots From Matlab **38**

List of Tables

2.1 Comparison of single-machine write bandwidth to instance local SSD and remote storage in Amazon EC2. [6] 11

List of Figures

1.1 Image processing example using a serverless architecture [3] 1

1.2 Popularity of the term "serverless" as reported by Google Trends [3]. Numbers represent search interest relative to the highest point on the chart. . . 4

1.3 Serverless platform architecture [3] 5

2.1 Benchmark of the serverless pools managing a concurrent and heterogeneous load [11]. It shows the performance difference between the standard Spark cluster and Databricks Serverless Pool when long-running tasks are assigned to the cluster with multiple short-running interactive queries. 10

2.2 PyWren architecture [6] 10

2.3 Prorated cost and performance for running 1TB sort benchmark while varying the number of Lambda workers and Redis shards. [6]. Each bar represents an execution time and each dot represents a cost. Tuples in x-axis contain the number of concurrent Lambda workers and Redis shards. . . . 12

2.4 Qubole Spark on Serverless [16] 13

3.1 Overview of the Flint architecture. 16

3.2 Spark workflow [7] 17

3.3 Shuffling and Dataflow of a two-stage job example 20

3.4 PySpark internals on standard Spark cluster [13]. Workers rely on Java/Scala Spark running on JVM 21

3.5 Timeout of Flint executor 22

3.6 Chained executors 23

4.1 Goldman Sachs Headquarters [15] 27
4.2 Query latency comparison. 28
4.3 Query cost comparison. 28
4.4 Concurrency tradeoff of q0 and q1. Each bar (with left y-axis) represents an execution time and each dot (with right y-axis) represents a cost 30
4.5 Concurrency tradeoff of q3. Each bar (with left y-axis) represents an execution time and each dot (with right y-axis) represents a cost 31

Chapter 1

Introduction

Serverless computing [3, 14] represents a natural next step of the "as a service" and resource sharing trends in cloud computing. Specifically, "function as a service" offerings such as AWS Lambda allow developers to write blocks of code with well-defined entry and exit points, delegating all aspects of execution to the cloud provider. Typically, these blocks of code are stateless, reading from and writing to various "state as a service" offerings (databases, queues, persistent stores, etc.).

Standard serverless deployments are characterized by asynchronous, loosely-coupled, and event-driven processes that touch relatively small amounts of data [5]. Consider a canonical example that Amazon describes: an image processing pipeline such that when the user uploads an image to a website, it is placed in an S3 bucket, which then triggers a Lambda to perform thumbnail generation. The Lambda may then enqueue a message that triggers further downstream processing. This use case is illustrated in Figure 1.1. Most serverless applications are user facing, even if users are not directly involved in the processing pipeline.

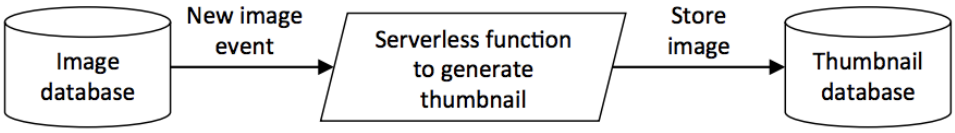


Figure 1.1: Image processing example using a serverless architecture [3]

This thesis explores serverless architectures for a completely different use case: large-scale analytical data processing by data scientists. Up until now, big data frameworks

such as Apache Spark and Apache Hadoop have been widely adopted and used by many data scientists for large-scale analytical data processing. Spark, which is a general-purpose engine that supports many different kinds of data processing tasks including SQL and streaming analysis, has become a crucial part of the big data stack as the adoption rate has been rapidly increasing. However, Spark and other big data frameworks need to be pre-installed on a cluster before they can be used for analytics.

There are multiple solutions available for data scientists/companies to adopt in order to get a Spark cluster ready and those solutions can be categorized into two classes. First, companies could deploy Spark in an on-premise environment. The IT team configures the multiple Spark clusters in their data center and manages them so that they can be used by data scientists in the company. However, this requires the company to be able to afford the data center as well as the resources in the IT team to manage the cluster since there can be various types of issues which could affect the availability and/or reliability of Spark jobs. Thus, it may not be a feasible solution for many relatively small companies and researchers with the limited resources.

Alternate approaches are based on employing a cloud service, which removes the need for a data center to set up the Spark clusters. One of such approaches is for each data scientist to run their own clusters on the set of virtual machines such as EC2 provided by the cloud service providers. The benefit of this approach is that users have full control over the cluster and do not have to rely on the IT team to make changes for quick iterations. However, this approach requires the data scientists to do the cluster configuration and management, which can be complicated tasks demanding significant efforts. The cluster management involves manual tasks such as monitoring the health of worker nodes, fixing/replacing underperforming nodes, rescaling the cluster as needed, and diagnosing the cluster when a variety of problems arises. This troubleshooting could require some knowledge about the internals of the big data framework and could take days and weeks especially for data scientists without a strong engineering background.

Another cloud-based approach is for each data scientist to use a managed big data frameworks such as Elastic Map Reduce (EMR). EMR provides a cluster of EC2 instances with big data frameworks such as Spark and Pig installed and configured on them. It also provides highly available slave nodes by automatically replacing unhealthy nodes with new nodes so that users do not have to monitor the health status and replace the node manually. Thus, EMR can greatly reduce the burden of cluster management. Moreover, when the cluster is created with a job, it can be automatically terminated after a job is finished, and this could significantly save costs. However, although it made big data frameworks much easier to use, data scientists are still obligated to choose the details of the managed cluster including the instance type and pricing model. Also, with services like

EMR, it is still required for the end users to decide ahead of time the number of machines to be used. Thus, even when running a simple *ad hoc* query, the management overhead is unavoidable. Another disadvantage is that a lot of time is wasted in cluster initialization/rescaling/teardown. It was observed that it usually takes more than 5 minutes for the cluster initialization, and this could be overkill for many *ad hoc* queries running only for a short amount of time.

Third cloud-based approach is using shared clusters on the cloud. Similar to on-premise approach, the IT team creates multiple clusters and manages them. The difference is that virtual machines (EC2 instances) are used instead of physical hardwares owned by the company. Since data scientists use the shared clusters, they can focus more on data science delegating devOps to the IT team. However, this means that companies should allocate their human resources for the cloud management tasks including configuration, monitoring, rescaling as well as troubleshooting. This may not be feasible for many small startups with the limited resources.

An alternative is to use tools such as Databrick's Unified Analytics Platform to manage clusters in the cloud. Here, we describe standard cluster and not Databricks Serverless, which will be described in the Related Work section. Similar to AWS EMR, Databricks runs Spark application under user's cloud account (AWS, Azure), but it also provides some useful features. First, it makes the collaboration of multiple users easier by supporting Interactive UI features including notebooks and dashboard. Moreover, the platform supports multiple optimization techniques including data skipping and automatic caching. However, even if cluster standup and teardown were completely automated (and instantaneous, let's even say), the fact remains that the organization pays for cluster instances for the entire time the cluster is up; charges accumulate even when the cluster is idle. For large organizations, this is less of an issue as there is more predictable aggregate load from teams of data scientists, but for smaller organizations, usage is far more sporadic and difficult to estimate a priori.

1.1 Serverless Architecture

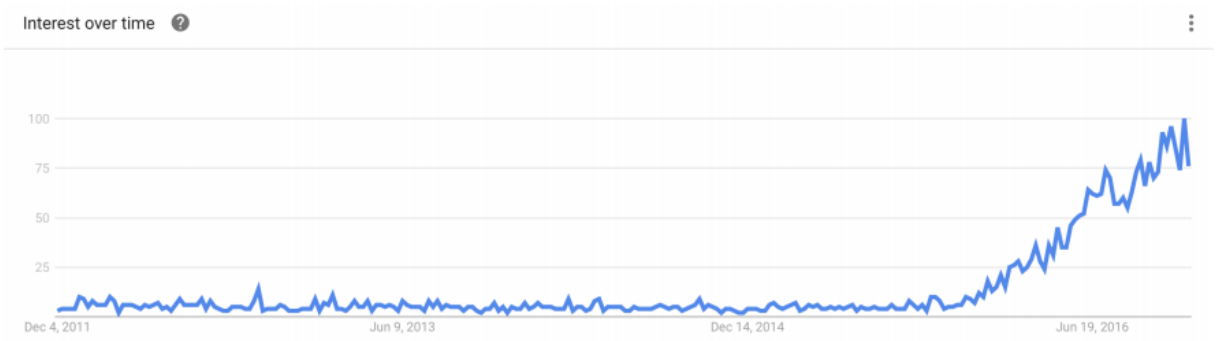


Figure 1.2: Popularity of the term "serverless" as reported by Google Trends [3]. Numbers represent search interest relative to the highest point on the chart.

Serverless computing refers to the programming model and architecture where small code snippets, specified by developers, are executed in the cloud without any control over the resources [3]. With the recent trends towards the microservice architecture, serverless computing has been gaining popularity and this is illustrated in Figure 1.2.

Figure 1.3 depicts the architecture of the serverless platform. The platform is triggered by an event (e.g HTTP) which contains the user code to be executed in the cloud. Events are first stored in the queue and scheduled to be processed. Then the dispatcher either selects the existing container of the serverless function or create a new container. After an event is received, the container executes the user-defined function and returns the response which is processed by the dispatcher and returned to the user.

Serverless architecture provides the abstraction layer so that any management concerns related to servers and infrastructures are delegated to cloud providers such as AWS. Thus, from the user's perspective, servers do not exist and this greatly reduces the operational management overheads. Users could concentrate more on the business logic of the modularized code instead of availability, fault-tolerance, and scalability.

However, there are also drawbacks of serverless computing. First, most serverless platforms currently have several limitations including the memory size (e.g 3008 MB for AWS) and execution time limitation. Also, given that serverless compute functions are stateless, it may be difficult for some applications to be transformed to adapt serverless computing.

Moreover, there are other limitations such as limited tools for monitoring and debugging as well as the vendor lock-in problem.

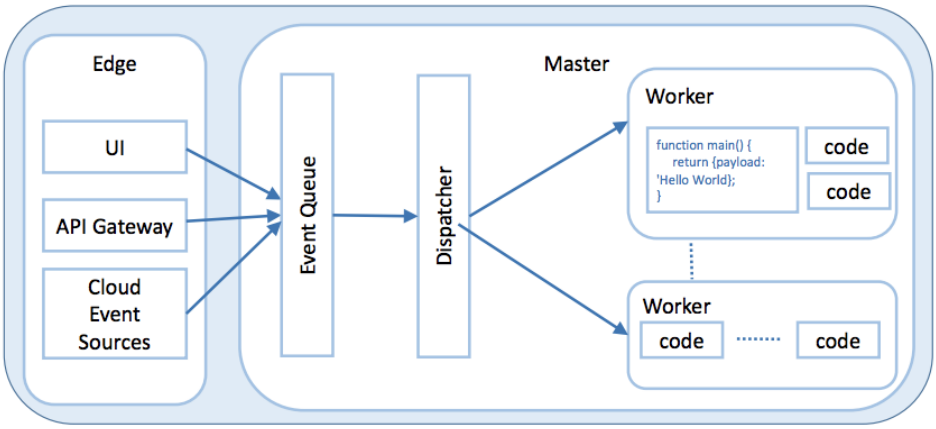


Figure 1.3: Serverless platform architecture [3]

1.2 Serverless Analytics

In this thesis, we describe Flint, a prototype Spark execution engine that is completely implemented using AWS Lambda and other services. One key feature is that we realize a pure pay-as-you-go cost model providing extreme elasticity, in that there are zero costs for idle capacity. With Flint, the data scientist can transparently use PySpark without needing an actual Spark cluster, instead paying only for the cost of running individual programs. Another key feature is the simplicity: Flint provides an easy interface for data scientists to use Spark for data analytics, simply by writing a Spark function to be deployed to the serverless backend without worrying about the cluster management overhead.

The primary contribution of our work is a demonstration that it is indeed possible to build an analytical data processing framework using a serverless architecture. Critically, we accomplish this using cloud infrastructure that has no idle costs. It is straightforward to see how workers performing simple "map" operations can execute inside Lambda functions. Physical plans that require data shuffling, however, are more complicated: Flint takes advantage of distributed message queues to handle shuffling of intermediate data, in effect offloading data movement to yet another cloud service.

Our vision is to provide the data scientist with an experience that is indistinguishable from "standard" Spark. The only difference is that the user supplies configuration data to use the Flint serverless backend for execution. In this context, we explore system performance tradeoffs in terms of query latency, cost, etc.

Currently, Flint is built on AWS, primarily using Lambda and other services. All input data to an analytical query is assumed to reside in an S3 bucket, and we assume that results are written to another S3 bucket or materialized on the client machine. The AWS platform was selected because it remains the most mature of the alternatives, but in principle, Flint can be re-targeted as other cloud providers have similar offerings.

One major design goal of Flint is to provide a truly pay-as-you-go cost model with no costs for idle capacity. This requires a bit of explanation: as a concrete example, Amazon Relational Database Service (RDS) requires the user to pay for database instances (per hour). This is *not* pay as you go because there are ongoing costs even when the system is idle. Therefore, this means that one obvious implementation of using RDS to manage intermediate data would violate our design goal. In general, we cannot rely on any persistent or long-running daemons.

Note that this is a challenging, but also worthwhile, design goal. In a cloud-based environment, there are a limited number of options for Spark analytics as described in the previous sections. We believe that serverless analytics with pay-as-you-go pricing is compelling, particularly for *ad hoc* analytics and exploratory data analysis. This is exactly what our Flint prototype provides.

Chapter 2

Related Work

Given all the challenges mentioned in the preceding chapter, there have been previous approaches to serverless analytics. In each of the sections below, we describe each solution in detail. Iris and PyWren are frameworks built from scratch on top of serverless compute functions and persistent storage. Amazon Athena supports serverless SQL experience using Presto, the distributed engine for query execution. Databricks Serverless provides the serverless experience of Spark to end users by using automatically managed pools of cloud resources. Finally, Qubole Spark on Lambda provides Spark on the serverless platform similar to Flint, but using the different approach of porting existing Spark executors infrastructure, whereas Flint uses from-scratch implementation.

2.1 Iris

The origins of Flint can be traced back to a course project called Iris at the University of Waterloo in the Fall semester of 2016. Iris is a distributed computation framework, supporting a subset of Spark API, built on top of AWS Lambda, S3, and other services.

Iris has a scheduler responsible for resource management, the web interface providing a monitoring tool that can be used to check the status of jobs, and the front-end library and shell for optimizations such as pipelining as well as the application programming interface. Iris then uses stateless Lambda functions to process data in parallel with injected user code and uses an external storage for shuffling. The number of concurrent invocation increases with the number of input splits up to the maximum value allowed for concurrency, and in this way, Iris exploits the extreme elasticity and scalability provided by the serverless

model. The performance was comparable to EMR Spark, especially for embarrassingly parallel tasks without intense data shuffling. Also, there was a big cost advantage back then since AWS EMR used per hour billing instead of per second billing, introduced in 2017. Since Iris supported Javascript API as well, in-browser data analytics backed by serverless backend without any servers was possible. *Ad hoc* data analysis could be as easy as writing a couple of lines of Javascript code to be executed in each remote container and clicking a button that triggers distributed processing on the cloud.

2.2 Amazon Athena & Amazon Redshift Spectrum

Amazon provides two data analytics services that are worth discussing: Amazon Athena (announced November 2016) and Amazon Redshift Spectrum (announced July 2017). Both are targeted at more traditional data warehousing applications and only support SQL queries, as opposed to a general-purpose computing platform like Spark. Athena offers a pay-as-you-go, per-query pricing with zero idle costs, similar to Flint, but under the covers it uses the Presto distributed SQL engine for query execution, so it is not built on top of the serverless platform. Redshift Spectrum is best described as a connector that supports querying over S3 data; the customer still pays for the cost of running the instances that comprise the Redshift cluster itself (i.e., per hour charge, even when idle).

2.3 Databricks Serverless

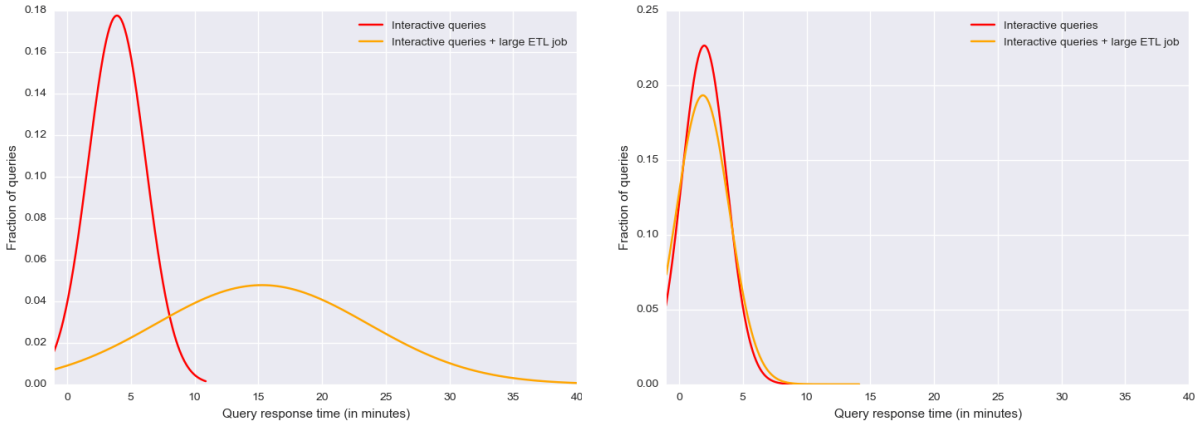
In June 2017, Databricks announced a serverless product [11]. This is best described as a more flexible resource manager: administrators define a "serverless pool" that elastically scales up and down. Prior to Databricks Serverless, other cloud managers, including Yarn, and the cloud services, including EMR, suffered from the various problems such as low utilization of resources and complex configurations.

Previously, multiple users of the same organization individually requested resources from the resource manager such as Yarn. However, this required users to predict the correct amount of resources needed and it is a challenging task that can be accomplished only after a lot of trial and error. Since underestimating the required resources leads to poor performance or even failure of Spark jobs, many users pick a number that is large enough to guarantee the successful completion of the jobs. As a result, the amounts of resources are chosen to be higher than needed, leading to a waste of resources. As resources

allocated to one user by the resource manager cannot be shared by another user, the waste from many different users accumulate for an organization and became more problematic.

Databricks Serverless instead provides a pool of auto-scaled resources. Since the resources can be shared among users in the organization, the resources can be more efficiently organized. It also provides autoscaling of the compute resources as well as the storage resources. Autoscaling of compute resources is based on Spark tasks in the queue, and Databricks claimed that this Spark-native approach resulted in the better utilization of resources compared to traditional resource managers using the coarse-grained autoscaling. Furthermore, having enough disk space is important for jobs to finish without out-of-memory error since the disk space is used for shuffling and disk spilling. Databricks Serverless provides the autoscaling of storage resources by automatically provisioning additional EBS volumes as the local storage of the instances becomes full. Also, serverless pools embeds preemption and fault isolation into Spark allowing users inside the pool to be isolated while sharing the resource pool: it prevents a single job from monopolizing all the resources which will slow down all other jobs, and also prevents a faulty code from bringing down the entire cluster. This can be helpful for optimizing the performance as well as increasing the availability of the system. As an example, consider a case where long-running ETL jobs are assigned to the cluster where multiple data scientists are running their short-running interactive queries. Then, in the case of standard Spark clusters, the average response times of interactive queries suddenly increase as interactive queries share the resources with the ETL jobs. On the other hand, Databricks pool provides the performance isolation by using proactive preemption minimizing the impact of the ETL job. This led to the difference in performance as shown in Figures 2.1a, 2.1b.

However, Databricks Serverless can be viewed as more convenient tooling around traditional Spark clusters, and is not serverless in the sense that we mean here as it is not built on top of the serverless platform although the goal is to also to provide the serverless experience of Spark. Since it still manages the cluster under the hood, it requires the end users to start the cluster and wait for it to become ready although the cluster startup is much faster with Databricks Serverless compared to other resource managers.



(a) 20 Users on Standard Cluster

(b) 20 Users on Serverless Pool

Figure 2.1: Benchmark of the serverless pools managing a concurrent and heterogeneous load [11]. It shows the performance difference between the standard Spark cluster and Databricks Serverless Pool when long-running tasks are assigned to the cluster with multiple short-running interactive queries.

2.4 PyWren

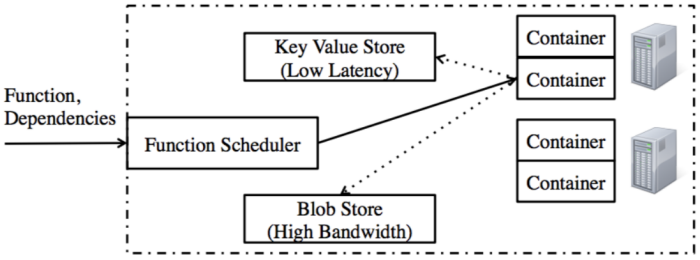


Figure 2.2: PyWren architecture [6]

In June 2017, Jonas et al. [6] introduced PyWren. PyWren is another project advocating a serverless execution model for analytics tasks, although unlike our effort PyWren does not attempt to target Spark or a specific analytics framework. Since Flint is a Spark execution

engine, it supports arbitrary RDD transformations; in contrast, PyWren examines only three classes of dataflow patterns: map-only, map + monolithic reduce, and MapReduce.

Storage Medium	**Write Speed (MB/s)**
SSD on c3.8xlarge	208.73
SSD on i2.8xlarge	460.36
4 SSDs on i2.8xlarge	1768.04
S3	501.13

Table 2.1: Comparison of single-machine write bandwidth to instance local SSD and remote storage in Amazon EC2. [6]

PyWren experimented with two different external storages for shuffling, S3 and Redis. The overall architecture of Pywren is shown in Figure 2.2

The paper first describes the use of S3 for shuffling. According to the authors, S3 worked well for data shuffling of 83.68M Amazon reviews using 333 partitions, only 17% slower than Spark not including the startup time of the Spark cluster. However, with a shuffle-intensive workload with the Terasort [12] algorithm on 1TB input, 6250000 intermediate files were created for shuffling of 2500 tasks at each stage. Although S3 provides enough I/O bandwidth to Lambda, sustaining such high request throughput became the bottleneck. In order to solve this problem, Redis cluster was employed since it provides low latency and high throughput storage. As a result, redis handled the job of sorting 1TB quite well as shown in Figure 2.3. Redis was also used to implement parameter-server [8, 2] style applications in PyWren. However, Redis cluster is not pay-as-you-go in the AWS pricing model (i.e, It has idle costs involving the long-running daemons requiring users to plan and set up before usage).

Throughout the paper, the authors demonstrated the promise of a data processing system built with stateless functions and remote storage. PyWren was used to successfully process data in many applications in the variety of fields including computational imaging, scientific instrument design, solar physics, and object recognition. Moreover, the paper stated that the current trends, where the gap between network and storage I/O bandwidth is decreasing, benefits of co-locating computation and data have diminished. Their benchmarks showing the I/O throughput versus the network throughput is shown in Table 2.1.

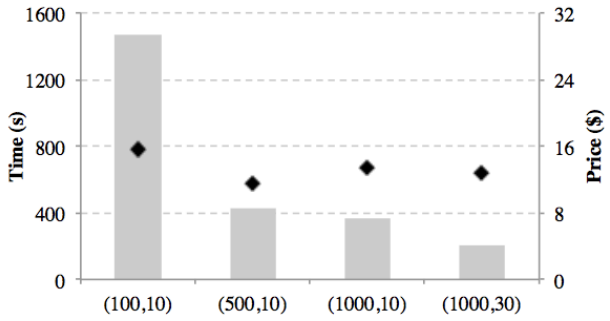


Figure 2.3: Prorated cost and performance for running 1TB sort benchmark while varying the number of Lambda workers and Redis shards. [6]. Each bar represents an execution time and each dot represents a cost. Tuples in x-axis contain the number of concurrent Lambda workers and Redis shards.

2.5 Qubole Spark on Lambda

In November 2017, Qubole announced an implementation of Spark on AWS Lambda [16]. This effort shares the same goals as Flint, but with several important differences which will be discussed in this section as well as in the discussion section.

In Qubole Spark, the driver runs on an EC2 instance with a security group configured to allow incoming connections from executors on Lambda. Lambda functions and EC2 instance are configured to be on the same VPC. Figure 2.4 illustrates the architecture of Qubole's Spark on Lambda.

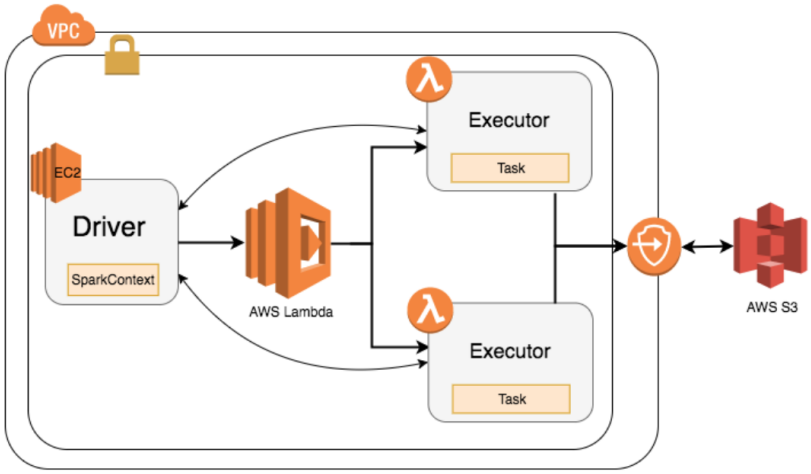


Figure 2.4: Qubole Spark on Serverless [16]

Since the maximum deployment package size of Lambda is small (50 MB), initially, Python Lambda container without Spark is launched. Then, for new containers, Spark libraries are downloaded from a compressed S3 package to `/tmp`, disk storage allocated to Lambda function. The installed archive is then extracted and the Spark Executor is started with Java command line. Qubole reports that this process is fairly slow and it takes around 2 minutes. However, when the container is reused, the installation step is skipped and the startup time reduces to 4 seconds. Since the size of disk space provided is 50 MB, Qubole skimmed Spark, Hadoop, and Hive distribution (jars) to create a smaller package that fits the disk of Lambda functions.

Qubole implemented a scheduler for managing Lambda executors to be used for the Spark job executions. Since there is a 5 minutes limitation for the execution duration, a scheduler keeps track of the status of executors and do not assign tasks to executors that are older than the threshold. Currently, this threshold is set to 4 minutes by default.

Qubole uses S3 for shuffling data. Internally Hadoop file system is used and this makes it easy to replace S3 with other external data source such as Amazon Elastic File System for a shuffle store.

2.6 Summary

Serverless computing, in general, is an emerging computing paradigm and previous work has mostly focused on examining system-level issues in *building* serverless infrastructure [9]

as opposed to designing applications. Indeed, as Baldini et al. [3] write, the advantages of serverless architectures are most apparent for bursty, compute-intensive workloads and event-based processing pipelines. Data analytics represents a completely different workload and Flint opens up exploration in a completely different part of the application space.

Chapter 3

Architecture

3.1 Design

The overall architecture of Flint is shown in Figure 3.1. At a high-level, Spark tasks are executed in Amazon Web Services (AWS) Lambda, and intermediate data are held in Amazon's Simple Queue Service (SQS), which handle the persistence and the data shuffling necessary to implement many transformations. For each task of Spark jobs, the Flint scheduler in the driver invokes AWS Lambda function, which creates or reuses a container with Flint executor code bundle. The serialized task metadata is sent as an argument of Lambda function, and the Lambda container starts the Flint executor that retrieves the metadata and processes the Spark task.

To maximize compatibility with the broader Spark ecosystem, Flint reuses as many existing Spark components as possible. When a Spark job is submitted, the sequence of RDD transformations (i.e., the RDD lineage) is converted into a physical execution plan using the DAG Scheduler. The physical plan consists of a number of stages, and within each stage, there are a collection of tasks. The `Task Scheduler` is then responsible for coordinating the execution of these tasks. Spark provides pluggable execution engines via the `SchedulerBackend` interface: Spark by default comes with implementations for local mode, Spark on YARN and Mesos, etc. Flint provides a serverless implementation of `SchedulerBackend`; the workflow and the core concepts remain unchanged from standard Spark. The workflow is shown in Figure 3.2. The primary advantage of this design is that we can reuse existing Spark machinery for query planning and optimization, and Flint only needs to "know about" Spark execution stages and tasks in the physical plan. Spark also provides many different types of RDD transformations as well as the higher level libraries

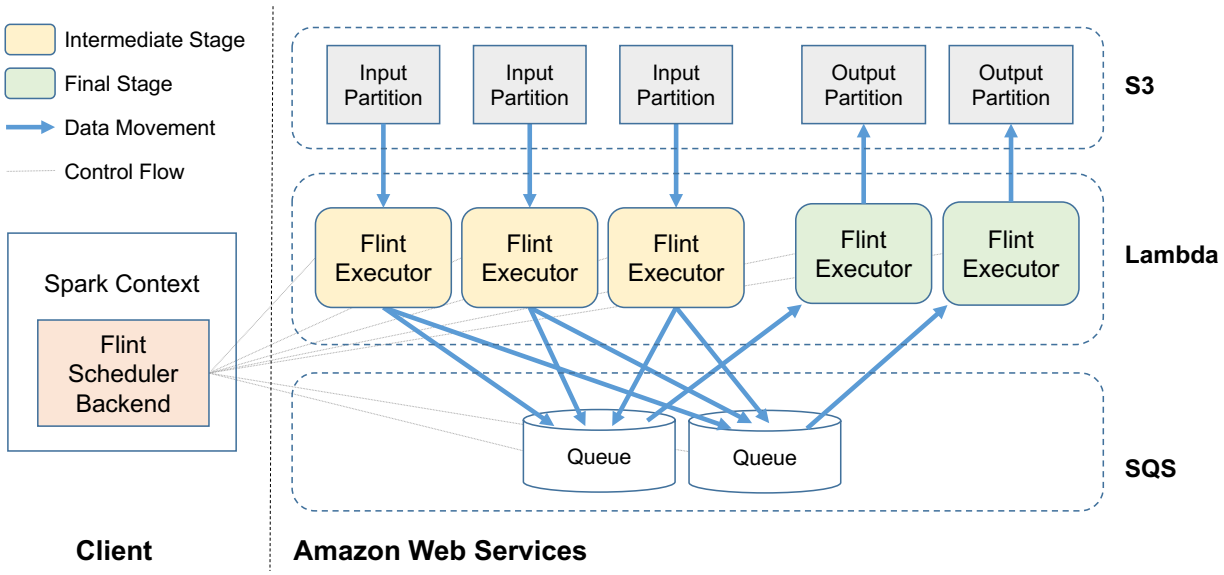


Figure 3.1: Overview of the Flint architecture.

including Spark SQL, MLlib (Machine Learning library), Spark Streaming and GraphX built on top of Spark engine. Reusing many of Spark components makes it much easier to support such high-level data analysis tools for data scientists.

The Flint `SchedulerBackend` (hereafter "scheduler"), which lives inside the Spark context on the client machine, is responsible for coordinating Flint executors to execute a particular physical plan. The Flint `SchedulerBackend` is created when SparkContext is initiated if parameters are set to use Flint engine at the backend. When it is instantiated, it first checks if the AWS Lambda function for Flint executor with the name provided by the configuration already exists and ready to be used. If the function is not registered, the scheduler registers Lambda function to be used by all the jobs it handles. The scheduler contains the pool of threads to manage lambda executions, and the size of the pool decides the number of the maximum concurrent lambda invocations. This number is read from the Spark properties. When there are idle threads in this pool, the scheduler requests and receives tasks from Spark's `Task Scheduler`. For each task, our scheduler implementation extracts and serializes the information that is needed by the Flint executors. This information includes the serialized code to execute, metadata about the relationship of this task to the entire physical plan, and metadata about where the executor reads its input and writes its output. When this serialization is complete, the scheduler uses the idle threads to asynchronously launch the Flint executors by invoking the registered AWS

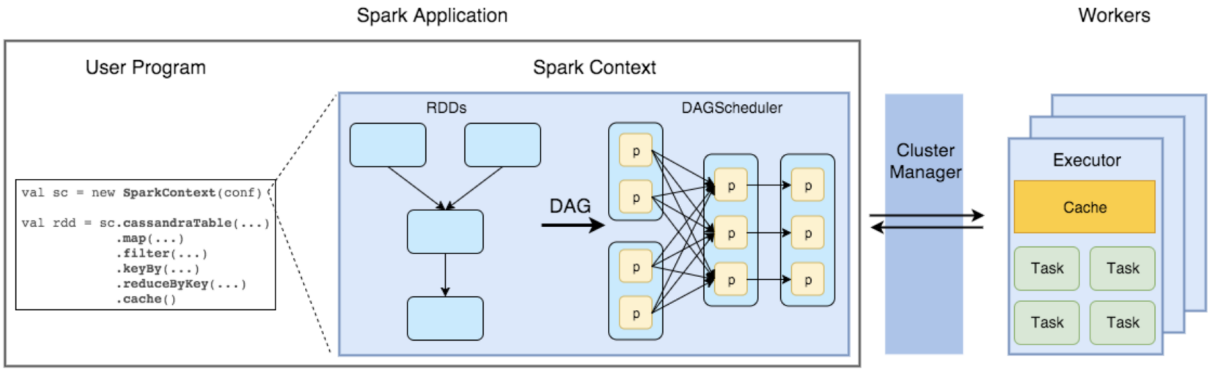


Figure 3.2: Spark workflow [7]

Lambda function, with the serialized task as part of the request. After a Flint executor has completed its task, the scheduler processes the response from an executor. Once all tasks of the current stage complete, executors for tasks of the next stage are launched, repeating until the entire physical plan has been executed.

3.2 Flint Executor

A Flint executor is a process running inside an Amazon Lambda function that executes a task in a Spark physical plan. When a newly registered AWS Lambda function is first invoked, a new container with resources is created with the code of a Flint executor. For subsequent invocations, new containers may or may not be created depending on if there are available idle containers. When a container stays in idle state for a while, it gets terminated and becomes unavailable for other invocations. Creating a new container with appropriate resources adds latency, but once a container is initiated, it is used over and over for other invocations for this job. Multiple jobs could also share the executors if the parameters are set appropriately to use the same Lambda function. However, this additional latency means that a cold-start problem exists even for a serverless model although it is negligible compared to server-oriented resource models.

Since the startup latency of a Lambda invocation is small once the function has been "warmed up" (i.e., a container is reused), each Lambda instance only processes a single task. This is different from standard Spark cluster implementations, where executors are long-running processes that span the lifetime of a Spark job. This design simplifies the communication requirement between an executor and a driver since the driver only has to

provide the task information to an executor when it is first started at the invocation and the executor only has to return the result of a task execution at the end. Additionally, executors are more likely to be less affected by the limitation of Lambda execution time of 5 minutes.

Once a Flint executor has started inside a container, it first deserializes the task information from the request arguments. Depending on how big an input task is, it may also read a part of metadata from S3 which will be further discussed in a later section. From the input partition metadata, the executor creates an input iterator to read from the appropriate input partition. For the first stage in a plan, this iterator will fetch a range of bytes from an S3 object. For most other stages, the input iterator will fetch from a designated SQS queue (discussed in detail below).

Once the input iterator is ready, it is passed as an argument to the deserialized function (i.e., code to execute) from the task; this yields the output iterator. If the task is in the final (result) stage of the execution plan, there are two possibilities: If the final action on the RDD is `saveAsTextFile`, outputs are materialized to another S3 bucket; otherwise, the results are materialized within the executor and passed back to the scheduler (for example, if the data scientist calls the `collect` action). However, if the size of a result is too large, this will fail due to the response size limit of Lambda functions. This can be solved by first uploading the results to S3 and return the key information to the driver. Then, the driver can use this information to read the result from S3.

3.3 Remote Storage

When a task is part of an intermediate stage, the execution plan requires the output to be shuffled so that all values for a single key are placed in the same partition. The shuffling is part of the physical plan created by Spark; the Flint executors simply execute the task, and thus are not explicitly aware of the actual RDD transformation (e.g., if the shuffling is part of `reduceByKey` or `join`, etc.). For Spark clusters, executors communicate with each other to achieve shuffling. An executor executing a reduce task may read remotely from multiple executors the results from the previous stage. However, since the execution time of a Lambda invocation has a limit of 300 seconds, it is not possible to guarantee that the Flint executors used from the previous stage are still alive to pass data to executors running tasks from the next stage. Moreover, the size of disk space provided for a Lambda function is too small to rely on. Thus, we need some external data store to deliver the intermediate output. Flint uses Amazon's Simple Queue Service (SQS) for this purpose as it is highly scalable and reliable with no idle cost; SQS is billed with the number of SQS

requests and the size of data transfer, which is free for data transferred between Amazon SQS and Amazon EC2 within a single region.

Once an executor of a task belonging to an intermediate stage has computed the output iterator, the hash partition function (or custom partition function if specified) is used to decide which partition each output object will be assigned to. The executor groups objects by the destination partition in memory. However, if memory usage becomes too high during this process, the executor flushes its in-memory buffers by creating a batch of SQS messages and sending them to the appropriate queue for each partition using the pool of dedicated threads. After all output data are sent to SQS queues, the executor terminates and returns a response containing a variety of diagnostic information (e.g., number of messages, SQS calls, etc.).

Once all tasks of the current stage are completed, executors for tasks in the next stage will be launched. These executors read from their corresponding SQS queues and aggregate data in memory. The aggregated results are passed to the iterator of the function associated with the task, as described earlier. Figure 3.3 illustrates the dataflow of a two-stage job example involving a shuffling stage. Since we are using in-memory data structure for aggregation, memory forms a bottleneck. Due to the complexities of implementing on-disk multi-pass aggregation algorithms in the Lambda environment, we currently address this problem by increasing the number of partitions such that we do not overflow memory. This solution appears to be adequate since it takes advantage of the elasticity provided by AWS Lambda. Once the executor is done reading the input data from a SQS queue, it deletes this queue.

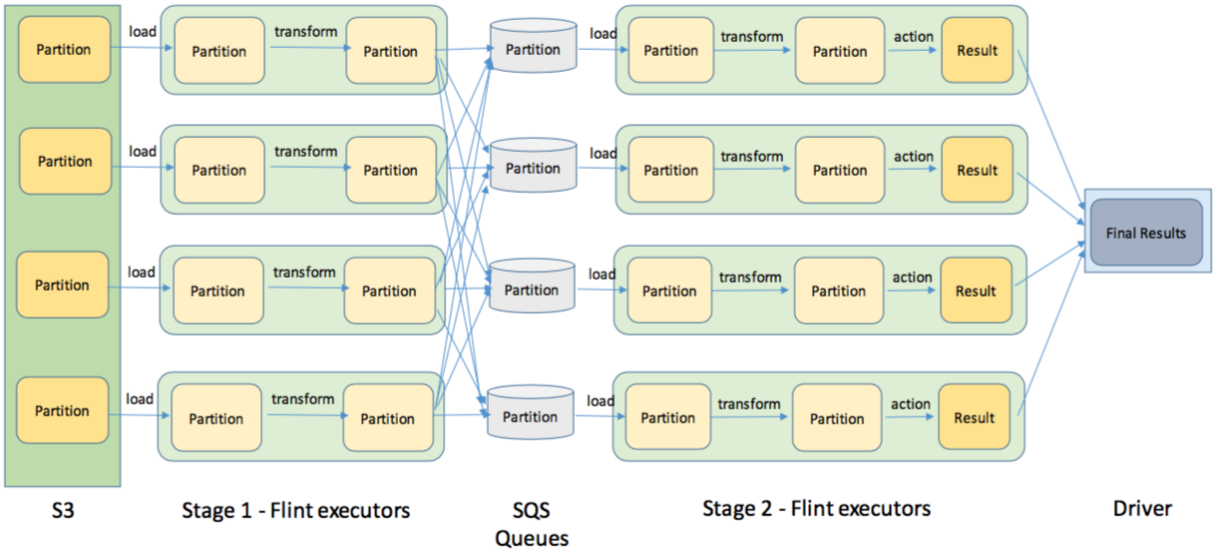


Figure 3.3: Shuffling and Dataflow of a two-stage job example

Queue initialization is performed by the scheduler. Before the execution of each stage, the scheduler creates queues for the necessary partitions. Partition metadata (i.e., the name of the source queue and the prefix of destination queues) are passed as part of the Lambda request.

If one or more executors fail to complete because of the unexpected error such as the network error, Flint achieves fault tolerance by simply re-executing only the executors that have failed if the interrupted task belongs to the final stage. Since the intermediate data of Flint are stored in the external data sources such as SQS, it is not required to re-execute the whole stage as in standard Spark clusters to compute the missing intermediate data split. However, if an executor failed to complete the task in the intermediate stage, some SQS queues may have the output from this failed executor. In this case, we cannot simply re-execute the executor. Thus, we currently re-execute the whole stage as a simple solution. However, we will explore other options such as using ‘message attributes‘ to mark which messages are from the failed executor and thus have to be ignored. If S3 is used in place of SQS for the intermediate data, we can simply re-execute the task (and not the whole stage) after deleting S3 spills of the failed executor, assuming executors do not share S3

destinations for the intermediate data.

3.4 Overcoming Lambda Limitations

The current Flint implementation supports PySpark, which counter-intuitively is easier to support than Scala Spark. The Flint `SchedulerBackend` on the client is implemented in Java, but the Flint executors in AWS Lambda are implemented in Python. This design addresses one of the biggest issues with AWS Lambda: the long cold startup time of function invocations. The first time that a Lambda is invoked (or after a period of inactivity), AWS needs to provision the appropriate runtime container. Some experiments [4] showed that compiled languages such as Java takes longer than dynamic languages such as Python for creating a container. Java/Scala Spark has more dependencies as well as more logic compared to PySpark since PySpark is more or less a python API to use the Spark framework.

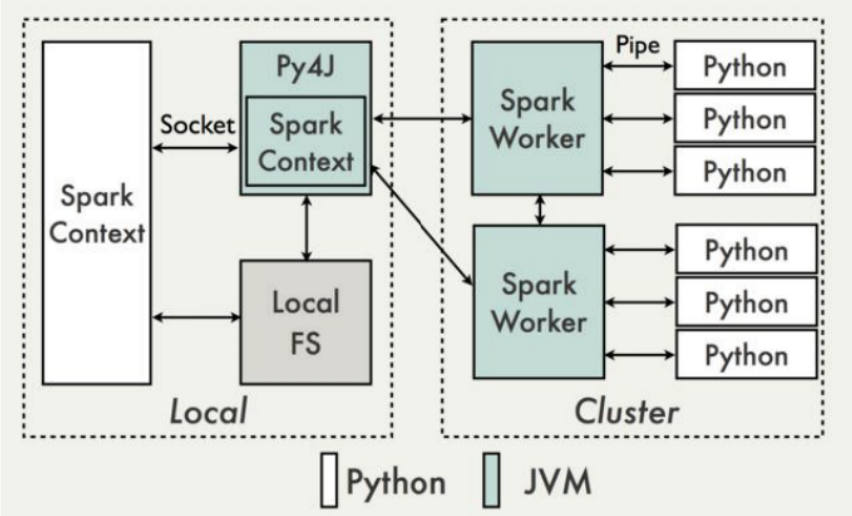


Figure 3.4: PySpark internals on standard Spark cluster [13]. Workers rely on Java/Scala Spark running on JVM

Consequently, in the default Spark executor implementation, PySpark relies on Spark for many functionalities such as shuffling and reading input. In order to run PySpark code, data (from S3) is first read in the JVM and then passed to a Python subprocess using pipes. This is illustrated in Figure 3.4. In Flint, we bypass this extra wrapper layer, and Python code is able to read from S3 directly. As we later show, this has

significant performance advantages. Another way to minimize the cold start problem is to keep Lambda function instances warm by periodically invoking them concurrently for a very short time. This approach is used by Serverless Framework [10]. However, with this solution, it is required to plan ahead the resource usage (i.e., how many executors? for how long?) and invoking many instances every period (exact time before a Lambda instance becomes cold is unknown). Even for a short time, this can be quite expensive. For these reasons, Flint is currently not employing this solution.

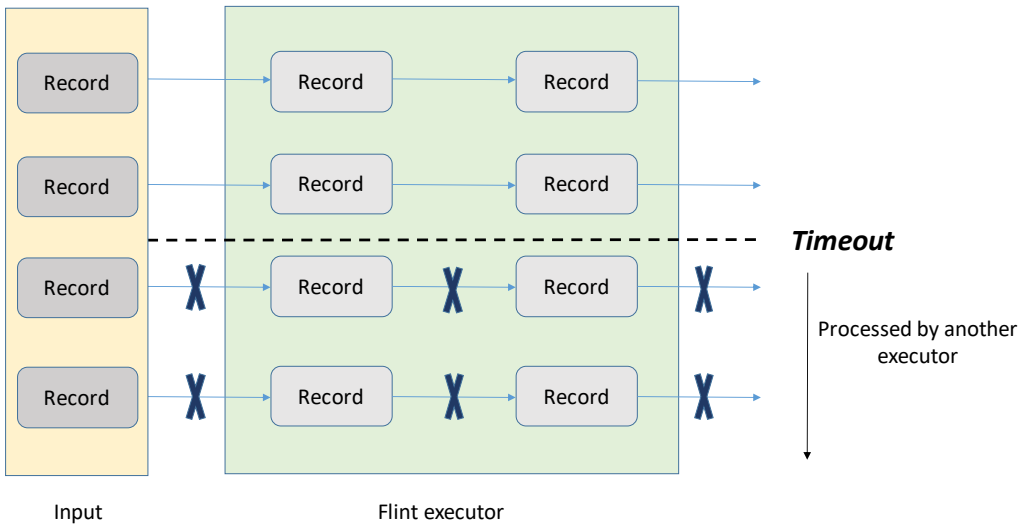


Figure 3.5: Timeout of Flint executor

Another limitation of AWS Lambda is that execution duration per request is capped at 300 seconds. This leads to the failure of long-running tasks. In order to avoid this problem, we chain executors: if the running time has almost reached the limit, the Flint executor stops ingesting new input records as shown in Figure 3.5. Since the transformation functions are pipelined and the input records are processed as they are fetched, the executor only has to take care of the batch of records that was most recently fetched. At this point, all previous batches of records were transformed and pushed to S3 or SQS, or merged to the result with the action function. Then, after all the records already in memory are processed, the current state, including how much of the input split has been read, is

serialized and returned to the scheduler, which launches a new Flint executor to continue processing the uncompleted input split from where the previous invocation left off. This process is illustrated in Figure 3.6. Since the function is already warm, the cost of using chained executors is reasonably low.

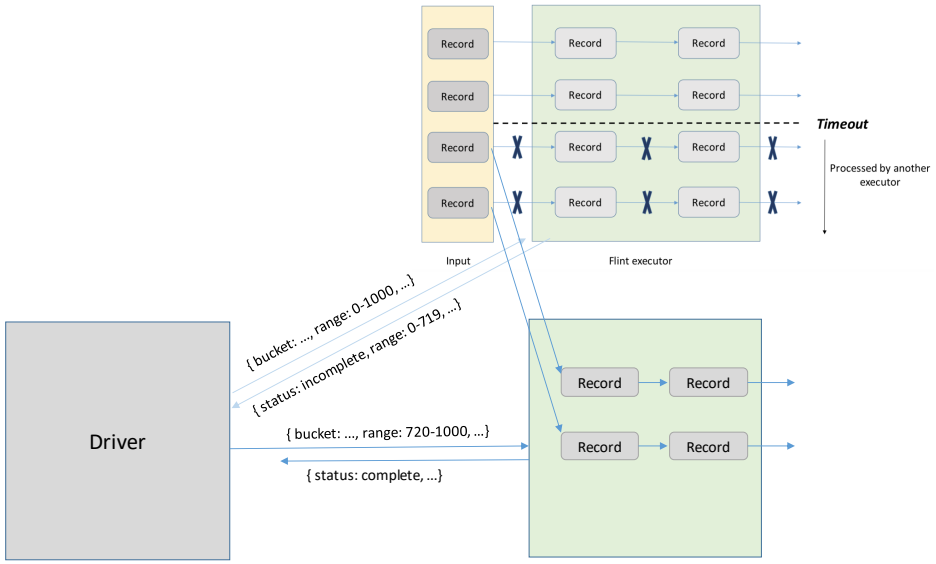


Figure 3.6: Chained executors

A third limitation of Lambda comes from a number of resource constraints. Each invocation is limited to a maximum memory allocation of 3008 MB. Thus, it is important for the Flint executor to carefully manage in-memory data. This constraint will be gradually eased as AWS increases the memory capacity. Since the CPU is allocated proportionally to the memory size, Flint executor uses the maximum memory of 3008 MB by default. There is also a limitation of 6 MB on the size of the request payload for an invocation. This payload is used to hold the serialized task data, which is typically much smaller. However, we are currently implementing a workaround for this size restriction by splitting the payload into smaller pieces. These can be uploaded to S3, and the scheduler can direct the Lambda functions to fetch the relevant data to complete initialization. The response payload also has a limitation of 6 MB. Since it is not recommended to collect a large amount of data, this is of less concern. However, for the completeness, Flint will support

collecting large data by executors first writing to S3 buckets and the scheduler reading them in the driver.

Chapter 4

Experimental Evaluation

4.1 Flint Executor Performance

We evaluated PySpark on Flint by comparing its performance with a Spark cluster running on the Databricks Unified Analytics Platform. The entire cluster comprises 11 m4.2xlarge instances (one driver and ten workers), with a total of 80 vCores (Amazon's computation unit) of processing capacity. For AWS Lambda, we allocated the maximum amount of memory possible, which is 3008 MB. The reason for using the maximum memory is to optimize the latency since AWS Lambda allocates CPU power proportional to the memory size. The developer can also configure the maximum number of concurrent invocations. As shown in our experiment, the performance gets better while the cost stays constant as the number of concurrent invocation increases. However, it is expected that the high request rate of reading S3 buckets can become a bottleneck when there are too many concurrent executors. We set the maximum number of concurrent invocations to 80 to match the Spark cluster, under the assumption that one lambda invocation roughly uses one vCore. AWS is not completely transparent about the instances running AWS Lambda; documentation refers to a "general purpose Amazon EC2 instance type, such as an M3 type" without additional details. Thus, this is the best that we can do to ensure that all conditions consume the same hardware resources. In all cases, we used the latest version of the Databricks runtime, which is based on Spark 2.3.

Our evaluations examined three different conditions: PySpark code running on Flint, PySpark code running on the Spark cluster, and equivalent Scala Spark code running on the Spark cluster. For the Spark cluster, we only measure query execution time (derived from cluster logs) and do not include startup costs of the cluster (around five minutes). This puts

Spark performance in the best possible light. We had separately examined Amazon EMR, which initializes clusters automatically per job—but for reasons unknown from available documentation, its performance (even excluding startup costs) was significantly worse than a Spark cluster we provisioned ourselves.

For evaluation, we considered a typical exploratory data analysis task described in a popular blog post by Todd Schneider [15]. The New York City Taxi & Limousine Commission has released a detailed historical dataset covering approximately 1.3 billion taxi trips in the city from January 2009 through June 2016. The entire dataset is stored on S3 and is around 215 GB. Each record contains information about pick-up and drop-off dates/time, trip distance, payment type, tip amount, etc. Inspired by Schneider's blog post, we evaluated the following queries, some of which replicated his analyses:

Q0. Line count. In this query, we simply counted the number of lines in the dataset. This evaluates the raw I/O performance of S3 under our experimental conditions.

Q1. Taxi drop-offs at the dedicated driveway, Hudson River Greenway, of Goldman Sachs headquarters at 200 West St as shown in Figure 4.1. This query filters by geo coordinates and aggregates by hour, as follows:

```
arr = src.map(lambda x: x.split(',')) \
.filter(lambda x: inside(x, goldman)) \
.map(lambda x: (get_hour(x), 1)) \
.reduceByKey(add, 30) \
.collect()
```

This is exactly the query issued in PySpark to both Flint and the Spark cluster. The Scala Spark condition evaluates exactly the same query, except in Scala. Note that Flint is able to support UDFs transparently.

For brevity, we omit code for the following queries and instead provide a concise verbal description. The details of the queries are available in the appendix.

Q2. Same query as above, but for Citigroup headquarters, at 388 Greenwich St.

Q3. Goldman Sachs taxi drop-offs with tips greater than $10. Who are the generous tippers?

Q4. Cash vs. credit card payments. This query computes the proportion of rides paid for using credit cards, aggregated monthly across the dataset.

Q5. Yellow taxi vs. green taxi. These two are the well-known taxi types in New York city. The query computes the number of taxi rides for each type, aggregated by month.

Q6. Effect of precipitation on taxi trips, i.e., do people take taxi more when it rains? This query analyzes the relationship between the amount of precipitation and taxi rides by aggregating average daily trips for different bucket ranges of precipitation per day.

Figure 4.1: Goldman Sachs Headquarters [15]

Figure 4.2 and Figure 4.3 report latency (in seconds) and the estimated cost of each query (in USD) under the three different experimental conditions. We report averages over five trials (after warm-up). Estimated costs for Spark and PySpark are computed as the query latency multiplied by the per-second cost of the cluster. For Flint, we used logging information to compute the execution duration of the AWS Lambdas and the associated SQS costs.

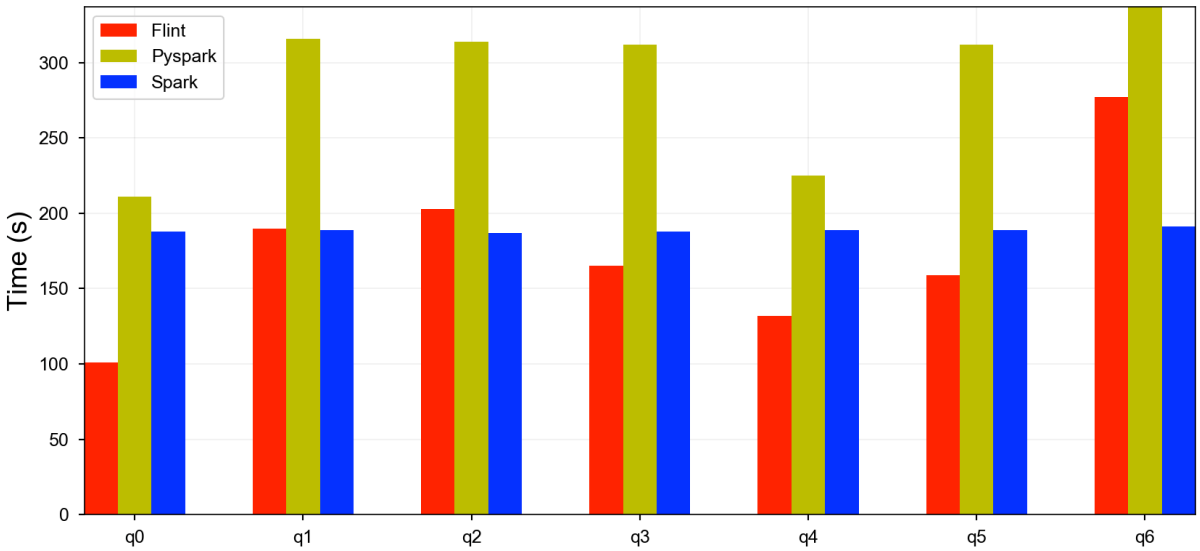


Figure 4.2: Query latency comparison.

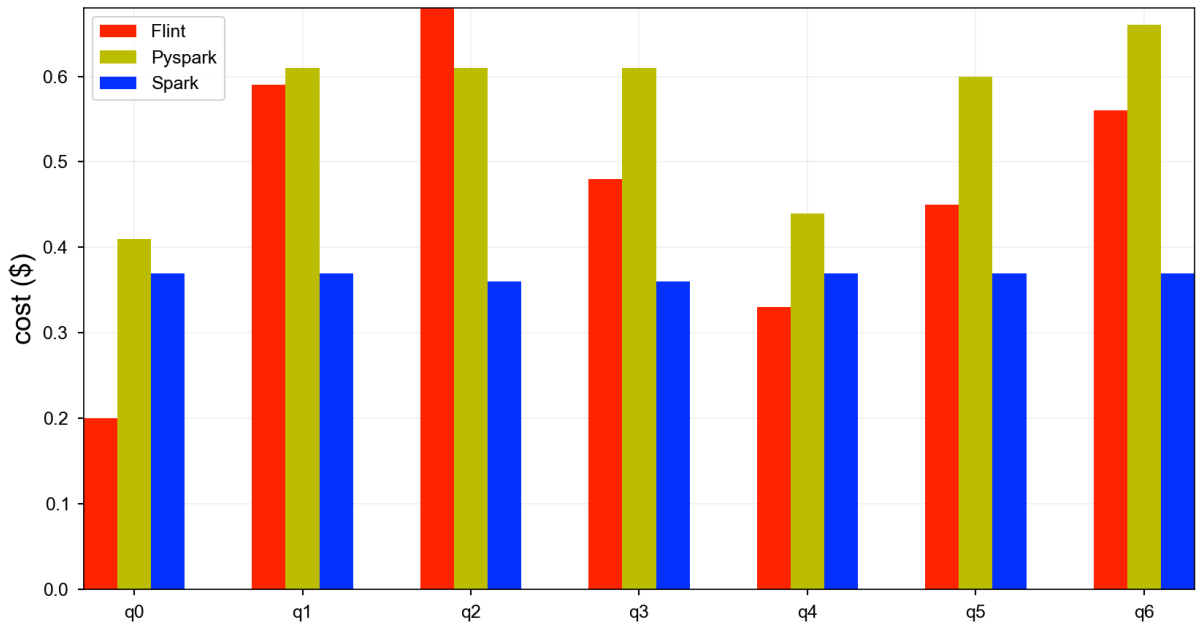


Figure 4.3: Query cost comparison.

We find that latency is roughly the same for all queries on Spark and appears to be dominated by the cost of reading from S3. This is perhaps not surprising since none of our test queries are shuffle intensive, as the number of intermediate groups is modest. Interestingly, for some queries, Flint is actually faster than Spark. The explanation for this can be found in Q0, which simply counts lines in the dataset and represents a test of read throughput from S3. Evidently, the Python library that we use (boto) achieves better throughput than the library that Spark uses to read from S3. This is confirmed via microbenchmarks that isolate read throughput from a single EC2 instance. In our queries, the performance of Flint appears dependent on the number of intermediate groups, and this variability makes sense as we are offloading data movement to SQS. PySpark is much slower than Flint because every input record passes from the JVM to the Python interpreter, which adds a significant amount of overhead. In terms of query costs, Flint is in general more expensive than Spark, even for queries with similar running times (Flint has additional SQS costs). Although a direct cost conversion between Lambda and dedicated EC2 instances is difficult (the actual instance type and the multiplexing mechanism are both unknown), it makes sense that Lambda has a higher per-unit resource cost, which corresponds to the price we pay for on-demand invocation, elasticity, etc.

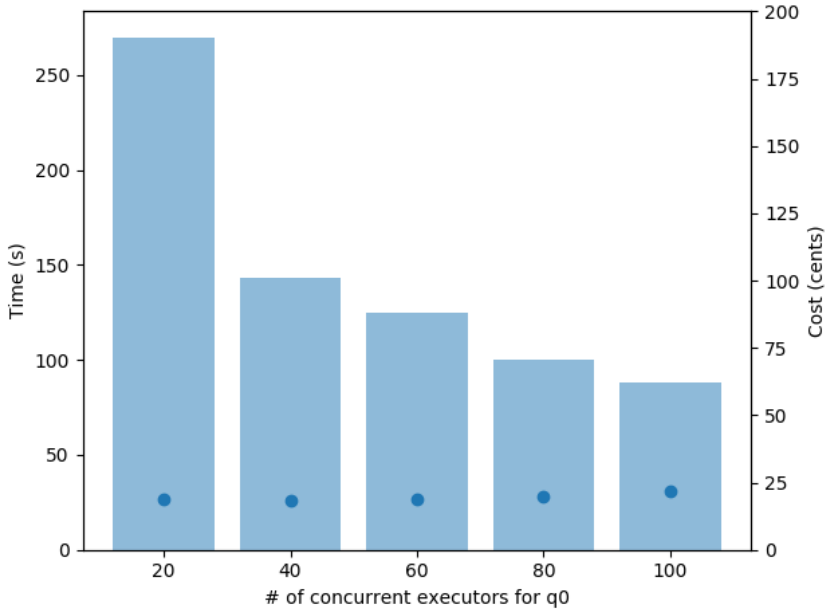
For the above reasons, it is difficult to obtain a truly fair comparison between Flint and Spark. Nevertheless, our experiments show that serverless analytics is feasible, though a broader range of queries is needed to tease apart performance and cost differences—for example, large complex joins, iterative algorithms, etc. However, results do suggest that data shuffling is a potential area for future improvement.

4.2 Latency/Cost Tradeoffs

Additionally, we evaluated the tradeoff of concurrency between the latency and cost. Q0, Q1, and Q3 described in the previous section were used for evaluation. Throughout the experiment, the number of total executors used is the same for each run, but the maximum number of executors running in parallel varies.

Figures 4.4, 4.5 show the result. As shown in the Figure, the execution time decreases as the number of concurrent executors increases. However, the cost stays relatively constant compared to the execution time. This shows one of advantages of the serverless model and that data scientists can achieve better performance with more parallelism at relatively small additional costs.

Q0: Simply count the number of taxi trips



Q1: Hourly aggregation of taxi drop-offs at the Goldman Sachs headquarters

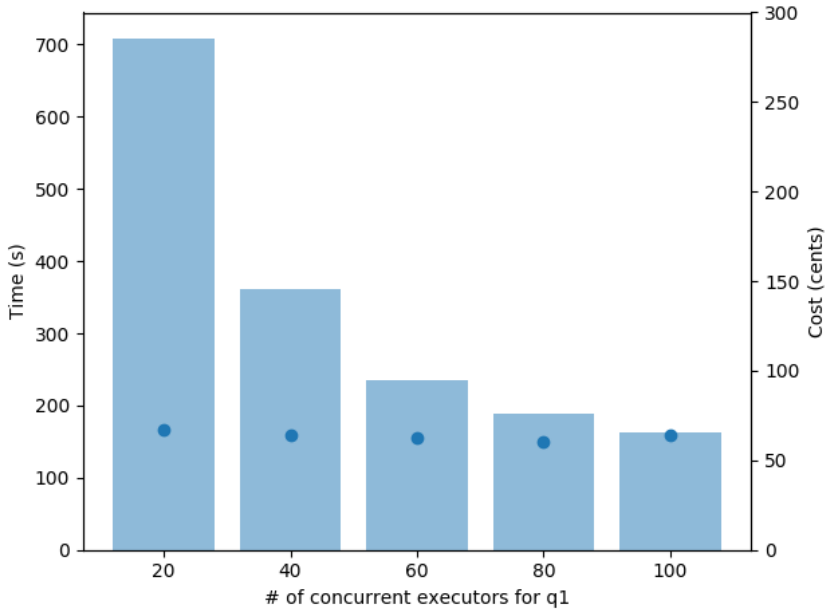


Figure 4.4: Concurrency tradeoff of q0 and q1. Each bar (with left y-axis) represents an execution time and each dot (with right y-axis) represents a cost

Q3: Goldman Sachs taxi drop-offs with tips greater than $10. Who are the generous tippers?

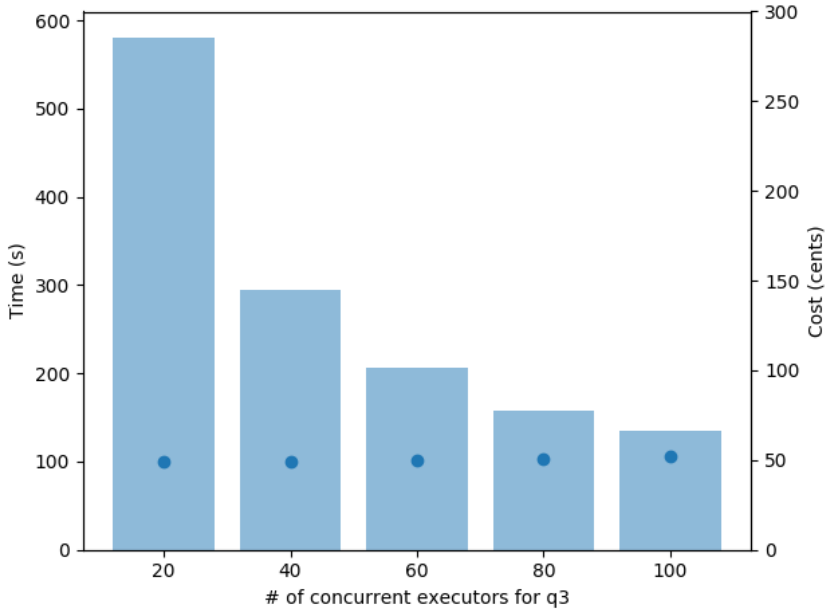


Figure 4.5: Concurrency tradeoff of q3. Each bar (with left y-axis) represents an execution time and each dot (with right y-axis) represents a cost

4.3 Discussion

As shown in the previous chapter, the performance of Flint is comparable to Spark on the standard cluster. Flint is easy to use without having to worry about the backend cluster and it also provides extreme elasticity inherited from the serverless model. Consequently, Flint seems to be a viable option for data scientists running sparse *ad hoc* queries, especially when the cluster startup time is taken into account. However, it may not be an ideal solution for batch production jobs, where jobs are running for a long time, because of the additional cost for using serverless compute functions.

As briefly mentioned earlier, there are several important differences between Qubole Spark on Lambda and Flint. Qubole attempted to port the existing Spark executor infrastructure onto AWS Lambda, whereas Flint is a from-scratch implementation. As a result, we are better able to optimize for the unique execution environment of Lambda. For example, Qubole reports executor startup time to be around two minutes in the cold start case.

For many short-running *ad hoc* queries, this startup time can be non-negligible. Furthermore, Amazon Lambda comes with several limitations and porting Spark executor could make it more difficult to deal with them. One of the limitations is five minutes execution duration time, and Qubole states that this is one of the areas that their implementation can be improved for scalability and stability. The problem becomes more apparent as the Qubole's executor runs one task after another until it is done as in a standard Spark cluster. Their current solution is to stop registering after four minutes. However, this means that the executor is vulnerable to long-running tasks requiring extra care in choosing the split size. Also, since executors on Lambda run until it times out waiting for the next task, resources can be wasted as an executor is just waiting even when there are no more tasks for it to run for a while. Moreover, Qubole's Spark on Lambda currently requires the driver to run on EC2 instance with the VPN setting as well as the security setting allowing executors on Lambda to communicate with the driver. On the other hand, the communication model between the driver and an executor in Flint is very simple consisting of invocation and response only, and thus Flint driver can be running anywhere without worrying about the network and security settings.

In addition, Qubole's implementation uses S3 directly for the shuffling of intermediate data, which differs from our SQS-based shuffle. Using S3 allows Qubole's executors to remain more faithful to Spark, but we believe that the I/O patterns are not a good fit for S3.

Chapter 5

Conclusion

5.1 Future Work

There are a number of future directions that we are actively exploring. We have not been able to conduct an experimental evaluation between Qubole's implementation and Flint, but the design choice of using S3 vs. SQS for data shuffling should be examined in detail. Each service has its strengths and weaknesses, and we can imagine hybrid techniques that exploit the strengths of both.

Robustness is an issue that we have not explored at all in this work, although to some extent the point of serverless designs is to offload these problems to the cloud provider. Executor failures can be overcome by retries, but another issue is the at-least-once message semantics of SQS. Under typical operating circumstances, SQS messages are only delivered once, but AWS documentation explicitly acknowledges the possibility of duplicate messages. We believe that this issue can be overcome with sequence ids to deduplicate message batches, as the exact physical plan is known ahead of time.

Another ongoing effort is to ensure that higher-level Spark libraries (e.g., MLlib, SparkSQL, etc.) work with Flint. To the extent that `SchedulerBackend` provides a clean abstraction, in theory, everything should work transparently. However, as every developer knows, abstractions are always leaky, with hidden dependencies. We are pushing the limits of Flint by iteratively expanding the scope of libraries and features we use.

5.2 Summary

To conclude, we note that Flint is interesting in two different ways: First, we show that big data analytics is feasible using a serverless architecture, and that we can coordinate the data shuffling associated with analytical queries in a restrictive execution environment. Second, there are compelling reasons to prefer using our execution engine over Spark's default, particularly for *ad hoc* analytics and exploratory data analysis: the tradeoff is a bit of performance for elasticity in a pure pay-as-you-go cost model. Thus, Flint appears to be both architecturally interesting as well as potentially useful.

References

[1] Spark configuration. https://spark.apache.org/docs/2.3.0/configuration.html, 2017.

[2] Martn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In *arXiv:1706.03178v1*, 2017.

[4] Yan Cui. aws lambda compare coldstart time with different languages, memory and code sizes. https://theburningmonk.com/2017/06/aws-lambda-compare-coldstart-time-with-different-languages-memory-and-code-sizes/, 2017.

[5] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, Colorado, 2016.

[6] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[7] Anton Kirillov. Apache spark: core concepts, architecture and internals. https://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/, 2016.

[8] Mu Li, David G. Anderson, Jun Woo Park Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. 2014.

[9] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.

[10] Gonalo Neves. Keeping functions warm - how to fix aws lambda cold start issues. https://serverless.com/blog/keep-your-lambdas-warm/, 2017.

[11] Greg Owen, Eric Liang, Prakash Chockalingam, and Srinath Shankar. Databricks Serverless: Next generation resource management for Apache Spark. https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html, 2017.

[12] Owen OMalley. Terabyte sort on apache hadoop. http://sortbenchmark.org/YahooHadoop.pdf, 2008.

[13] Joshua Rosen and Sean Owen. Pyspark internals. https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals, 2016.

[14] Neil Savage. Going serverless. *Communications of the ACM*, 61(2):15–16, 2018.

[15] Todd W. Schneider. Analyzing 1.1 billion NYC taxi and Uber trips, with a vengeance. http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/, 2015.

[16] Venkat Sowrirajan, Bharath Bhushan, and Mayank Ahuja. Qubole announces Apache Spark on AWS Lambda. http://www.qubole.com/blog/spark-on-aws-lambda/, 2017.

APPENDICES

Appendix A

Evaluation queries

Q0. Line count. In this query, we simply counted the number of lines in the dataset. This evaluates the raw I/O performance of S3 under our experimental conditions.

```
cnt = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.count()
```

Q1. Taxi drop-offs at the dedicated driveway, Hudson River Greenway, of Goldman Sachs headquarters at 200 West St. This query filters by geo coordinates and aggregates by hour, as follows:

```
arr = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.filter(lambda x: inside(x, goldman)) \
.map(lambda x: (getHour(x), 1)) \
.reduceByKey(add, 24) \
.collect()
```

Q2. Same query as above, but for Citigroup headquarters, at 388 Greenwich St.

```
arr = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.filter(lambda x: inside(x, citi)) \
.map(lambda x: (getHour(x), 1)) \
.reduceByKey(add, 24) \
.collect()
```

Q3. Goldman Sachs taxi drop-offs with tips greater than $10. Who are the generous tippers?

```
cnt = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.filter(lambda x: inside(x, goldman)) \
.filter(isGenerousTipper) \
.count()
```

Q4. Cash vs. credit card payments. This query computes the proportion of rides paid for using credit cards, aggregated monthly across the dataset.

```
arr = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.mapPartition(countPaymentTypes) \
.collect() \
```

Q5. Yellow taxi vs. green taxi. This query computes the number of different taxi rides, aggregated by month.

```
arr = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.mapPartition(countTaxiTypes) \
.collect()
```

Q6. Effect of precipitation on taxi trips, i.e., do people take taxi more when it rains? This query analyzes the relationship between the amount of precipitation and taxi rides by aggregating average daily trips for different amounts of precipitation.

```
arr = src.map(lambda x: x.split(',')) \
.filter(isValidTuple) \
.map(lambda x: (getPrecipitation(x), 1)) \
.reduceByKey(add, 20) \
.collect()
```