Carnegie Mellon University
Software Engineering Institute

Experiences with Deploying Mothra in Amazon Web Services (AWS)

Brad Powell
Dan Ruef
John Stogoski

July 2021

TECHNICAL REPORT

CMU/SEI-2021-TR-007
DOI: 10.1184/R1/16455651

CERT Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Homeland Security under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

DM21-0610

Table of Contents

Executive Summary

We at the Software Engineering Institute (SEI) were asked to investigate answers to the following questions:

- Can Mothra[1] be deployed in a cloud environment?
- Can that deployment work as effectively as Mothra does in an on-premises environment?
- How can cloud deployment be best accomplished to optimize Mothra's performance?

We researched methods for deploying Mothra and its related system components in the Amazon Web Services (AWS) GovCloud environment. As part of that work, we quickly determined that, when deployed in the cloud, Mothra could be easily installed and operated at speeds that clearly met user needs.

We then planned to determine how to significantly improve Mothra's query performance. To accomplish that goal, we planned the following activities:

- Implement multiple system designs in the SEI's hybrid prototyping environment.
- Modify configurations as test results are examined to address observed problems.
- Develop simulators to produce flow volumes that match those observed on production systems.
- Execute test plans to evaluate the data ingest process and representative query operations.
- Develop new code to optimize data read operations.
- Tune system services (e.g., Spark).

Analyzing the results of evaluating the feasibility of deploying Mothra on AWS GovCloud identified opportunities for improving the performance of systems in that environment. The results of this project are the confirmation of Mothra's successful integration into AWS GovCloud and a set of *levers* that can be used for tuning system services to specific data characteristics. Those levers include *file read parameters* and *desired file size* (thus the number of files) stored in a system repository.

To systematically determine the optimal settings for operating in the AWS GovCloud environment, we generated multiple Mothra repositories with different file scenarios and executed a series of tests using a range of parameter settings. For instance, we discovered that a scheduled roll-up operation of files reduces the processing time needed to load data from the AWS S3 bucket. Conversely, making the files too large has an adverse effect.

[1] Mothra is a large-scale data processing platform developed and maintained by the SEI for network security analysis.

To implement a system in the AWS GovCloud environment, we recommend a phased deployment approach that incorporates iterative testing with data at scale. By incrementally adding and verifying functionality at key points of the process, the deployment minimizes the occurrence of complex problems and the need for prolonged troubleshooting.

Findings from this research can be used to inform a path forward for those implementing systems in the AWS GovCloud environment. In particular, our experiences deploying Mothra in AWS GovCloud highlight the importance of the following aspects of deployment:

1. Select the correct instance types.
2. Add capacity/services in the US-West Region.
3. Use infrastructure-as-code methods for quickly rebuilding components.
4. Use extensive logging and system metrics.
5. Ensure staff are sufficiently skilled to support engineering and operations.

We used these approaches in our recent two-day effort to re-deploy the cluster and transfer the full Mothra test repository to the US-West Region. Initial results are very encouraging.

Abstract

The Mothra large-scale data processing platform can be deployed in the AWS GovCloud environment. The Software Engineering Institute (SEI) evaluation of this deployment shows that it meets (and even exceeds) the operating requirements of the on-premises Mothra deployment. This report describes (1) how an SEI team developed an at-scale prototype of the on-premises system to test the performance of Mothra in the cloud and (2) the approaches the team recommends for similar deployments.

1 Introduction

The goal of this project is to assess the viability of deploying and operating Mothra in AWS GovCloud. To make this assessment, we built prototypes of increasing capability that progressed toward target system performance. The prototype ingested billions of flow records per day with appropriate content distributed through the data and made that data available for analysis in an acceptable amount of time.

The larger the scale of the production system, the more probable issues will occur at various points in the data flow. Therefore, using the prototype, we planned to determine an optimal deployment configuration for AWS GovCloud that can accommodate a large-scale system. This plan is a complicated task due to the wide array of tuning options and the technical differences of AWS services.

For this research project, we used the SEI's hybrid prototyping environment to implement, test, and optimize the prototype system. We used the on-premises Ixia traffic generator to create a synthetic data stream that resulted in a sizable data repository within AWS. The observed scale of Mothra's current production sensing and data collection was used to provide throughput requirements and evaluate the ingest operation.

We identified representative queries and analysis operations and used them as the basis for user performance testing, given the required data load. We used the Spark-Bench tool to perform automated testing, ensuring the level of consistency needed to evaluate different system configurations. Our project involved multiple teams within the SEI Security Automation Directorate that collaborated to address code development, system engineering, and testing. We used DevOps techniques where appropriate and captured deployment information as infrastructure-as-code products.

This report is organized into four main sections:

- **System Overview** describes the prototype system and environment used to test the ability to use Mothra on AWS GovCloud effectively.
- **Test Plan** describes the activities conducted by the project team to test the prototype's effectiveness.
- **Results** covers several topics and provides detailed data that was captured during testing.
- **Recommendations** describes the recommendations that resulted from this project.

The report also includes an appendix that contains lessons learned from this project.

2 System Description

This section describes the prototype we used to test Mothra's deployment in AWS GovCloud. We provide a high-level view of the system (in Figure 1) and detailed descriptions of critical components of that system, including software, hardware, and data used. We designed the prototype to not only reflect the characteristics of on-premises Mothra, but also the huge amount of data it must handle.

Deploying systems such as Mothra in AWS provides the system and its operators with several benefits, including provision speed and flexibility. That flexibility leads to a significant learning curve for those deploying a system since they must understand and manage all the options available in the new environment. Operations experience takes time to develop, and sharing lessons learned is critical.

There are two main functional areas available in the Mothra system: Ingest and Query. The system must be able to consume data files at scale while maintaining the ability to retrieve and analyze the data.

2.1 Proposed Solution

To construct a prototype deployed on AWS that mimics the Mothra system in its on-premises state, we needed to build a solution that would recreate not only the system but also the data and its immense volume. We chose to use YAF[2] for capturing data and IPFIX (IP Flow Information Export) as the data format. This section describes the high-level solution as designed by the project team.

YAF processes packet data via live captures from an interface into bidirectional flows, then it exports those flows to IPFIX collecting processes in the IPFIX file format.

Files containing IPFIX flow records are transmitted to AWS. As the data files arrive in AWS, they are processed (i.e., packed) for storage and made available to streaming operations such as the Analysis Pipeline caches. A central data repository that contains all records is available for analysis with additional derived data sets making specific workflows more efficient.

Derived data sets can be created by stream processing at ingest or by periodic batch queries of the central repository. These smaller data sets enable analysts to quickly query for key indicators, such as an IP address or domain name. If warranted, subsequent queries can then be scoped and

[2] YAF (Yet Another Flowmeter) was originally intended to be an experimental implementation for tracking development in the Internet Engineering Task Force (IETF) IPFIX working group, specifically bidirectional flow representation, archival storage formats, and structured data export with Deep Packet Inspection. It is designed to perform acceptably as a flow sensor on any network where white-box flow collection with commodity hardware is appropriate. YAF can and should be used on specialty hardware when scalability and performance are of concern.

executed to retrieve the full flows. Additional derived data sets can be generated over time as analysts identify specific information they use frequently. This approach helps improve the user experience.

Interactive analysis of the full-flow repository is available via a notebook server. There must also be facilities to execute analyst-driven automated jobs. The system must support multiple users simultaneously without significant delays when a job is submitted.

The initial deployment should focus on providing the analyst with the best experience available in the desired timeframe. Fast query response time and alternative tools that both simplify and augment analyses are needed to support operational mission activities. The environment should provide methods for applying sets of indicators (e.g., IP addresses, filenames) and integrating data-enrichment feeds (e.g., GeoIP, ASN, and Scan.io data sets). Elasticsearch and Kibana are used for visualizations and quick indicator lookups.

2.2 Prototyped Solution

Figure 1 illustrates the prototype we developed. We deployed Mothra to Amazon Elastic Map Reduce (EMR) running Spark and backed by the EMR File System (EMRFS) with storage in Amazon S3. EMRFS is an implementation of HDFS that all Amazon EMR clusters use for reading and writing regular files from EMR directly to S3. EMRFS provides the convenience of storing persistent data in S3 for use with Hadoop while also providing features like consistent viewing, data encryption, and elasticity.

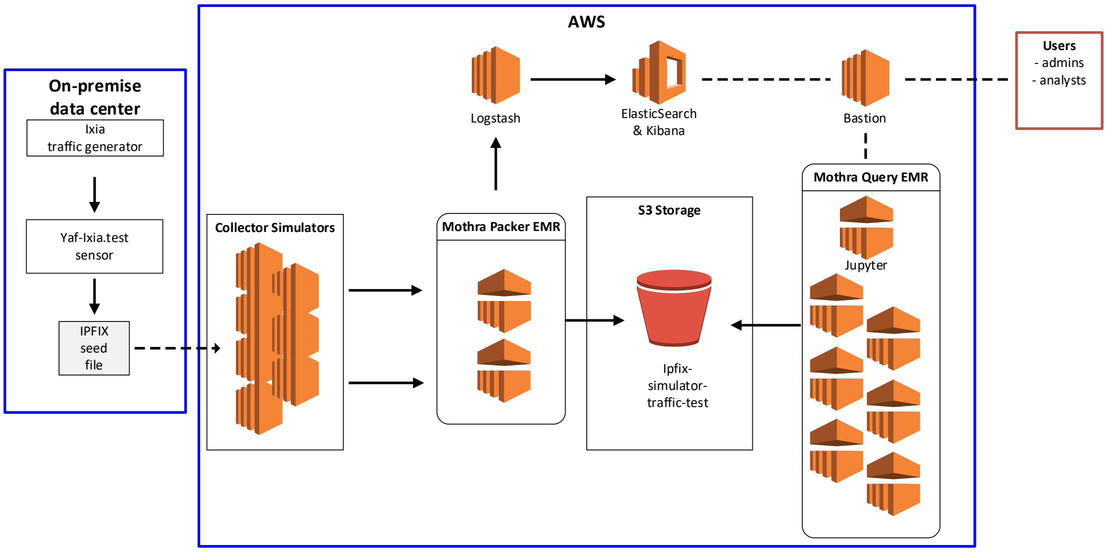


Figure 1: Prototyped Solution

To support several concurrent users, we configured EMR to use the YARN FairScheduler and enabled FairShare preemption. FairScheduler uses hierarchical queues. These queues are sibling queues when they have the same parent. The weight associated with a queue determines the amount of resources a queue deserves in relation to its sibling queues. This amount is known as *Steady FairShare*. The Steady FairShare is calculated at queue level and, for the root queue, is equivalent to all the cluster resources. These settings allow all users to be given a minimum share of resources on the cluster when needed unless certain high-priority queues are marked as non-preemptable.

JupyterHub on EMR allows user impersonation. All Spark jobs are submitted through the Livy API with a username attached. This allows better monitoring and auditing of cluster usage. User impersonation is supported by LDAP and PAM authentication and allows users to be mapped to groups/queues if necessary.

2.3 Data Requirements

The data requirements we used for our project were driven by the number of flows anticipated in Mothra's production environment. We investigated average flow and record size to estimate bandwidth and storage needs. We made calculations to understand the needed system capacity. The following are the resulting data requirements:

- 50 billion flows per day
- ~150 bytes per flow (non silkAppLabel = 0 flows are ~515 bytes)
- Estimated assumptions on the traffic profile
  - 80% of flows are between 7 a.m. – 7 p.m. factoring in time zones
  - Traffic flows are ~30% on weekends
- (5 days x 50B flows) + (2 days x 15B flows) = 280B flows/week or 1.2 trillion/month
- 150 bytes x 280B flows/week = 42 TB/week or 182 TB/month (uncompressed)
- Estimated peak traffic flow into the AWS environment
  - Considering there is ramp up in the morning and ramp down in the evening, divided 50% by 9 hrs = 2.8B flows/hr
  - 2.8B/60 mins = ~46M flows/min or 778k flows/sec
  - Assume 5-minute files from collectors; 100 collectors produce ~28,800 files per day (~1200 files/hr)

2.4 AWS Environment

EMR, is based on Apache Hadoop and requires three nodes to initiate a cluster. Master nodes run the resource manager and other services like Spark, Hive, Jupyter, and Ganglia. Three master nodes can, if desired, be configured to balance the load of these services.

Two core nodes are required to ensure basic functionality with HDFS and Hadoop. Task nodes can also be added to a cluster and allow MapReduce or Spark jobs to be processed without attached storage. Task nodes can scale elastically to a set maximum to meet resource demands on the cluster.

Table 1 and Table 2 illustrate the hardware and software resources needed for a Packer and Query EMR cluster.

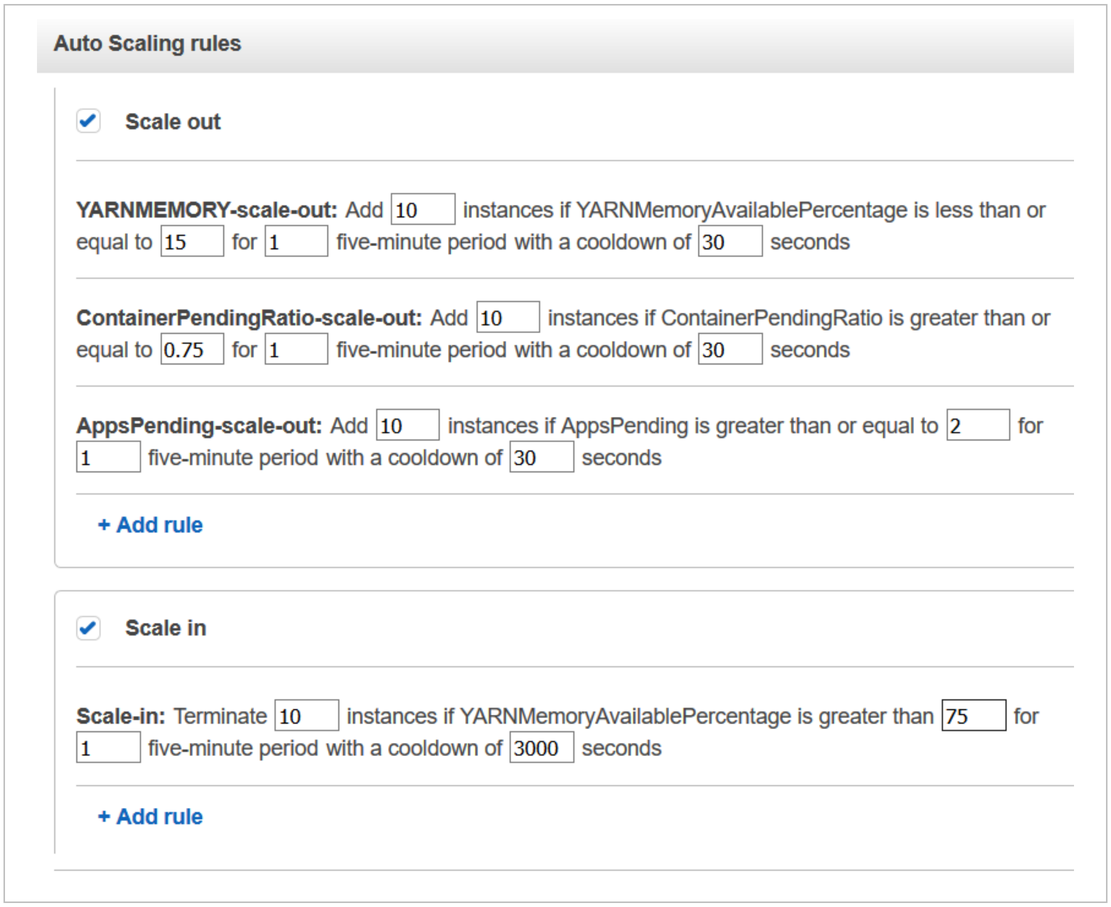
Table 1: Mothra Packer EMR Cluster

Hardware	**Software**
3 Master node – r5.4xlarge (16 vCore, 128 GiB memory) 2 Core nodes – r5.4xlarge (16 vCore, 128 GiB memory)	EMR-5.26.0 Amazon Hadoop 2.8.5 Spark 2.4.3 Livy 0.6.0 Ganglia 3.7.2

Table 2: Mothra Query EMR Cluster

Hardware	**Software**
1 Master node – r5.12xlarge (48 vCore, 384 GiB memory) 2 Core nodes – m5.2xlarge (8 vCore, 32 GiB memory) Elastic Task Instance Group – r5.4xlarge (16 vCore, 128 GiB memory) (minimum instances: 1, maximum instances: 40) (Auto Scaling On) Typical scale on us-gov-east-1 ~ 27 r5.4xlarge nodes €Typical scale on us-gov-west-1 ~ 167 r5.4xlarge nodes	EMR-5.26.0 Amazon Hadoop 2.8.5 Hive 2.3.5 Spark 2.4.3 JupyterHub 0.9.6 Livy 0.6.0 Ganglia 3.7.2

Auto-scaling rules (shown in Figure 2) allow the cluster to expand and contract elastically with the current resource demand. This flexibility ensures that resources are not sitting idle and incurring cost when they are not being used. Administrators can also configure these rules based on a schedule, such as at the beginning and end of a defined shift, so that resources are available for analysts and the expected performance of queries is met.



Auto Scaling rules

☑ **Scale out**

YARNMEMORY-scale-out: Add 10 instances if YARNMemoryAvailablePercentage is less than or equal to 15 for 1 five-minute period with a cooldown of 30 seconds

ContainerPendingRatio-scale-out: Add 10 instances if ContainerPendingRatio is greater than or equal to 0.75 for 1 five-minute period with a cooldown of 30 seconds

AppsPending-scale-out: Add 10 instances if AppsPending is greater than or equal to 2 for 1 five-minute period with a cooldown of 30 seconds

☑ **Scale in**

Scale-in: Terminate 10 instances if YARNMemoryAvailablePercentage is greater than 75 for 1 five-minute period with a cooldown of 3000 seconds

Figure 2: Auto-Scaling Rules

The Spark configurations we used are detailed in Table 3.

Table 3: Spark Configuration

Test Parameters	Values
spark.yarn.am.cores	2
spark.yarn.am.memory	14g
spark.scheduler.mode	FAIR
spark.driver.extraJavaOptions	-XX:+UseG1GC -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -XX:InitiatingHeapOccupancyPercent=35 -XX:OnOutOfMemoryError='kill -9 %p'
spark.submit.deployMode	client
spark.master	yarn
spark.executor.cores	2
spark.executor.memory	14g
spark.dynamicAllocation.enabled	true

2.5 Data Format and Structure

We made decisions about the data format and structure used for Mothra deployment on AWS to help maximize the efficiency of data transfer and storage.

The IETF[3] developed a standard for transferring IP flow data from exporters to collectors. This standard specifies the format for both data transfer and storage. A standard set of well-defined information elements and a method for transmitting and consuming system-specific elements enable multiple vendors to participate in a shared system. For instance, Cisco and VMware devices can export IPFIX data to a third-party collector, such as SiLK, for analysis.

IPFIX is a binary format that reduces the size of data and aids in processing when compared to text data formats. For example, an IP address is stored as a 32-bit integer, which is guaranteed to be smaller than a variable length string containing dots and digits (e.g., 1.2.3.4). By requiring that each data file contains a template that fully defines the included data records, IPFIX removes the need for a normalization step to fit every record into a fixed schema.

IPFIX also allows content-dependent nested values. These values reduce wasted storage space by eliminating the need for columns that are guaranteed to be empty when there are mutually exclusive subsets of data fields. The SEI network analysis and collection tool suite adheres to the IPFIX standard for its data, which means that data records are in their original format when stored in a Mothra repository.

Once the data is received, it must be stored using a method that supports the intended scale and use cases. Large-scale deployments must also consider disk I/O times and limit the size of reads where possible.

Mothra stores data in IPFIX files on disk and groups them according to date, collection source, and application information. Partitioning improves performance for queries involving date, collection source, and/or application, since entire files can be ignored because the system already “knows” that no records in the files meet the user’s filter criteria. The size of files is customizable and is optimized based on system resources. The filenames contain information used to determine which files must be opened based on the contents of the user’s query.

[3] Internet Engineering Task Force

```
.
└── ipfix-repeater                                   # <- S3 Bucket name
  └── 2019                                           # <- year
    └── 11                                           # <- month
      └── 14                                         # <- day
        └── v2
          └── eq=observationDomainId=17              # <- observation domain
              └── eq=silkAppLabel=53                 # <- silk app label
                └── FILE.ipfix                       #<- file name
```

Figure 3: Mothra Packer Partitioning Scheme

2.6 Sample S3 Key

We chose S3 as the storage solution. S3 offers a highly reliable, durable, and redundant storage solution. Spark matches partition selection criteria to S3 keys, which represent the location of IPFIX data matching those criteria. Figure 4 is a sample of an S3 key:

```
2019/11/14/v2/eq=observationDomainId=17/eq=silkAppLabel=53/20191114.00.9c568123-
def2-4694-9268-b08cd5854afc
```

Figure 4: Sample S3 Key

3 Test Description

To determine the optimal system deployment within AWS GovCloud, we conducted test scenarios. We used simulated data to create a repository of sufficient size to test the ingest and query operations and evaluate various configurations.

We created a collector simulator that we could scale horizontally to produce the desired number of flows per hour. The simulator used a seed file created by the Ixia traffic generator, modified record timestamps and IP addresses, and continuously output IPFIX files to be ingested by the system.

To examine the processing of data arriving from collectors and storing within AWS, we executed ingest tests. We executed query tests using a set of representative query operations to capture time durations.

We designed our evaluation of the AWS Mothra implementation to reflect typical, everyday use by analysts. Testing included using the IBM Spark-Bench benchmarking framework, which includes functionality to create custom tests. We created several custom tests to mimic how a user might query Mothra. We conducted these query tests using a variety of parameters, from storage solutions (e.g., S3 vs. HDFS), various Spark configurations (e.g., executor memory size, total executors per node), and the methods of partitioning the data (e.g., targetSize, numSlices).

To facilitate comparison, we tested with a specific version of Mothra, which was established as a baseline. We tested newer versions of Mothra under the same conditions and compared them to the baseline as a measure of improvement.

3.1 Data Sets

We used multiple data sets for testing. (See Table 4.) Initially, we configured a sensor in the on-premises data center to send synthetic data to the AWS environment. The data was processed and stored in an S3 bucket. After we completed initial functional testing, we deployed the collector simulators to increase the data rate to ~6M flows per day. The results of this test must be interpreted based on the specific data set used.

The flow repository used for these automated tests originally did not contain any records with a silkAppLabel equal to zero. Given the partitioning scheme used for Mothra, grouping records by silkAppLabel at the last level, all these extra flows would end up in their own partition. That means that any query that uses a filter of silkAppLabel not equal to zero would perform similarly, even with significant silkApplabel equal to zero flows present.

Table 4: Data Sets

ipfix-repeater (S3 Bucket) – ongoing feed from 50 collector simulators Size = ~2.6T/day. Objects per day = 57,800 (24,000 with new Mothra Packer 1 file per hour partition 10/1) 5.7 Billion flows per day, 515 bytes per flow, No records with silkAppLabel = 0 Dates: 07/22/19 – 10/26/19, 10/30/19 – 11/07/19 > 500 billion flows in the full repository
ixia-packed-streaming-test (S3 Bucket) – feed from Ixia, 8 sensors in the SEI RPID env Size = highly variable, on the order of a few hundred GB/day Objects per day = highly variable, on the order of a few thousand per day Dates: 05/31/19 - 06/21/19, 06/24/19 - 08/09/19, 08/30/19, 09/03/19. 09/04/19, 09/16/19-10/08/19
ipfix-data-testing (S3 Bucket) – multiple days of ipfix-repeater data rolled into 1 file per day, per partition. Using Mothra rollup-day tool to combine files by size or timeframe. Dates: 9/29/19, 10/01/19
ipfix-repeater (S3 Bucket) – ongoing feed from 30 collector simulators (new seed file 11/8) Size = ~3TB/day. Objects per day = ~20,000 ~14 Billion flows per day, appropriate silkAppLabel distribution Dates: 11/08/19 – current (Adding up to 100 simulators to get to ~50 billion flows/day) > 500 billion flows in the full repository
ipfix-repeater-west (S3 Bucket) – replicated data set from ipfix-repeater for testing in US-West Size = 213.2TB (as of 11/20). Objects = ~3.7 million (as of 11/20) Dates: 07/22/19 – current > 500 billion flows in the full repository

A seed file generated by the on-premises Ixia traffic generator includes simulated silkAppLabel distribution. The Spark configurations we used are detailed in Table 3.

Table 3 depicts a sample of traffic ratios for 12 hours of data from 30 collector simulators.

Table 3: Seed File Traffic Ratios

dns_flows	439149855	7.04%
smtp_flows	52324500	0.84%
https_flows	120922717	1.94
http_flows	506668515	8.12%
ftp_flows	52183493	0.84%
ssh_flows	52332037	0.84%
zero-flows	5015780948	80.39%
	6239362065	6239362065

3.2 Ingest Processing

The data input characteristics of Mothra in production include a constant stream of files during the course of the 24-hour day. The IPFIX files are transferred to the AWS environment as they arrive. The AWS components must be able to receive the files and partition (i.e., pack) the data at a speed that prevents backup and minimizes delay to the analyst.

The testing we performed on the prototype focused on the packing process and storage in AWS. The testing included three scenarios that corresponded to the need to scale up:

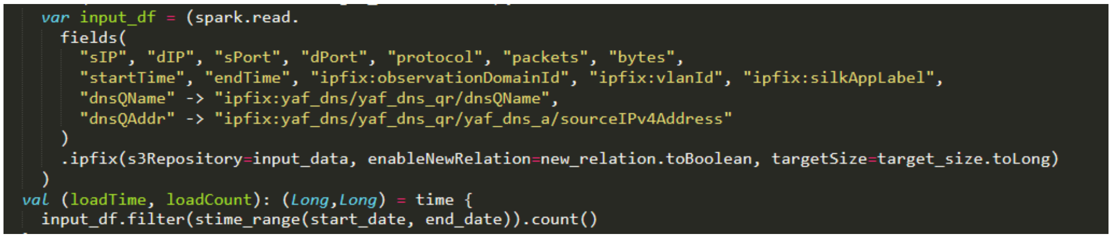
1. Synthetic data was generated by the Ixia/YAF-sensor solution residing within the SEI Data Center and sent to AWS.
2. Fifty collector simulators were deployed in AWS to generate ~6B flows per day using a generic enterprise traffic profile.
3. One hundred collector simulators were deployed in AWS to generate ~50B flows per day using a simulated traffic distribution.

Packer nodes are monitored to assess resource utilization to identify any points of constraint within the system.

3.3 Query Operations

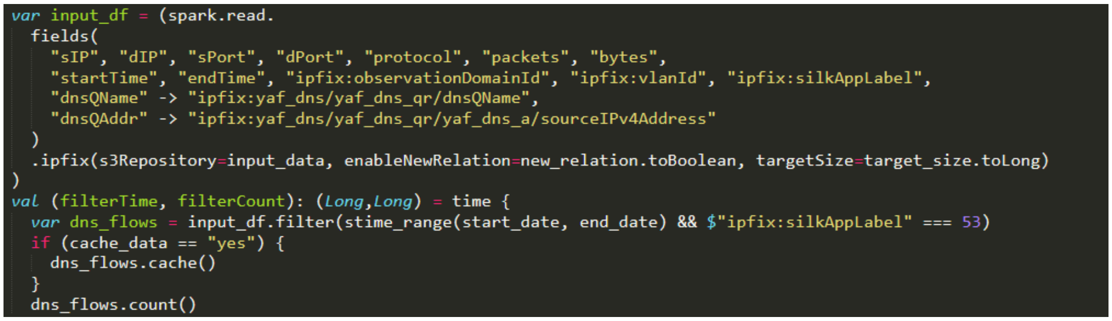
The following queries were written to assess performance of the system. We tested multiple date ranges for various numbers of flows as well as different values for targetSize and numSlices to find optimal query configurations.

1. **Load**: Count all flows for a given start and end date.
   The load query measures the time it takes to create a DataFrame in Spark, fill it, and count the data in a given timeframe.



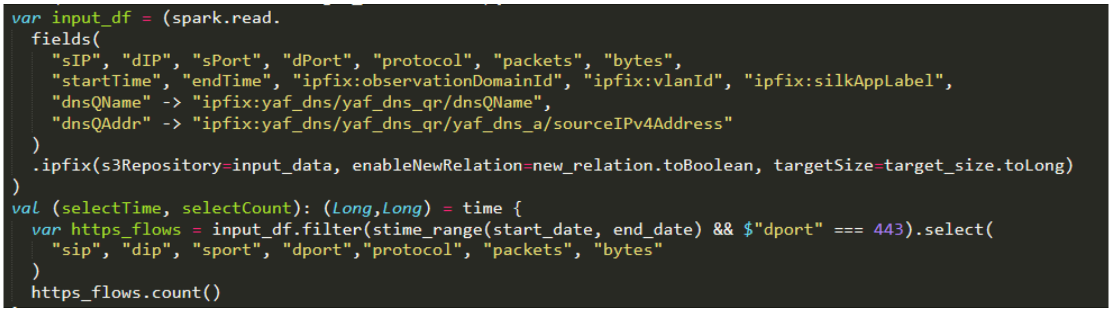
```
  var input_df = (spark.read.
    fields(
      "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
      "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
      "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
      "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
    )
    .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
  )
val (loadTime, loadCount): (Long,Long) = time {
  input_df.filter(stime_range(start_date, end_date)).count()
```

2. **Filter**: Filter for silkAppLabel = 53, and count all flows for a given start and end date. The filter query uses a Mothra Packer partition field (silkAppLabel), so when filtering on one of these fields, the Mothra code pulls only the files needed within that S3 object. Because Spark uses lazy evaluation (i.e., it performs operations only when needed), data is not loaded into a data frame until the subsequent filter command is run. Only data points that pass the Mothra partition-based filter are loaded (instead of every row in the repo), making this an important factor in performance.



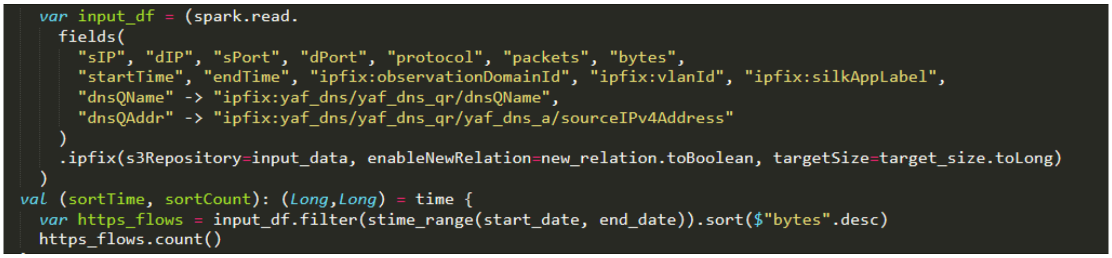
```
var input_df = (spark.read.
  fields(
    "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
    "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
    "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
    "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
  )
  .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
)
val (filterTime, filterCount): (Long,Long) = time {
  var dns_flows = input_df.filter(stime_range(start_date, end_date) && $"ipfix:silkAppLabel" === 53)
  if (cache_data == "yes") {
    dns_flows.cache()
  }
  dns_flows.count()
```

3. **Select**: Select fields from the DataFrame where dport = 443 for a given start and end date. The select query is a direct comparison to filter because it selects data from the repository where a specific dport is specified. This field is not being used as a Mothra packer partition, so it would need to look through the entire timeframe to find the matching flow records.



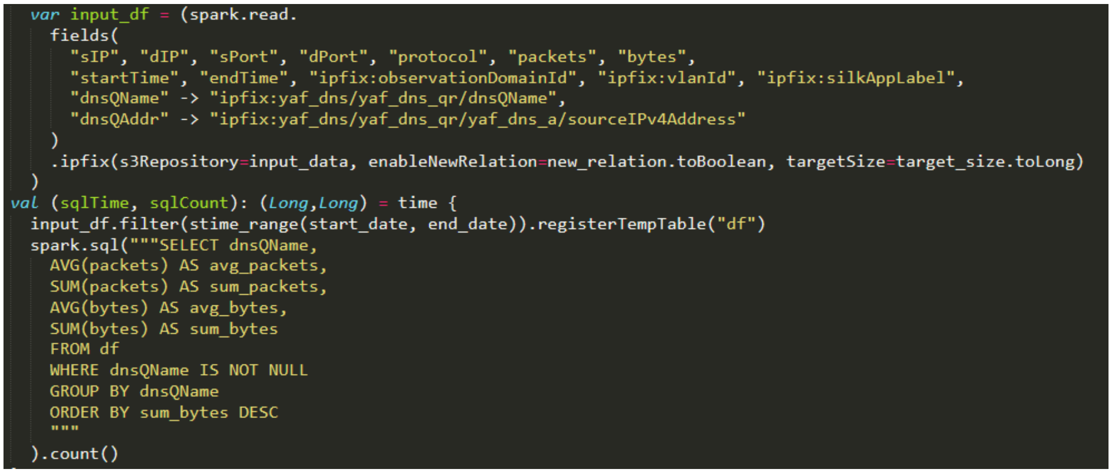
```
var input_df = (spark.read.
  fields(
    "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
    "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
    "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
    "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
  )
  .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
)
val (selectTime, selectCount): (Long,Long) = time {
  var https_flows = input_df.filter(stime_range(start_date, end_date) && $"dport" === 443).select(
    "sip", "dip", "sport", "dport","protocol", "packets", "bytes"
  )
  https_flows.count()
```

4. **Sort**: Sort by bytes descending, and count all flows for a given start and end date. Sort replicates an analyst looking for flows with the largest packet size.



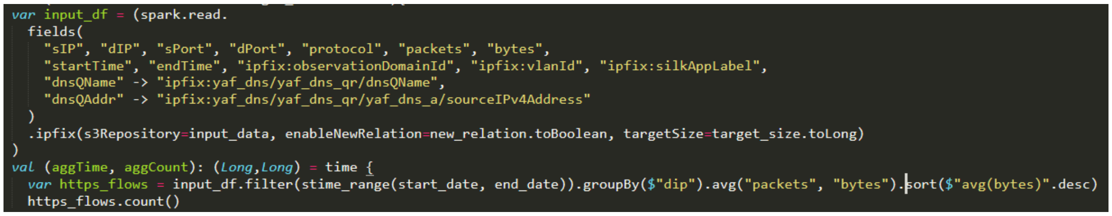
```
  var input_df = (spark.read.
    fields(
      "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
      "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
      "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
      "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
    )
    .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
  )
val (sortTime, sortCount): (Long,Long) = time {
  var https_flows = input_df.filter(stime_range(start_date, end_date)).sort($"bytes".desc)
  https_flows.count()
```

5. **SQL**: Execute a Spark SQL query where dnsQName is not null with group by, and order by for a given start and end date.
   dnsQName is not in the Mothra Partition.



```
  var input_df = (spark.read.
    fields(
      "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
      "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
      "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
      "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
    )
    .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
  )
val (sqlTime, sqlCount): (Long,Long) = time {
  input_df.filter(stime_range(start_date, end_date)).registerTempTable("df")
  spark.sql("""SELECT dnsQName,
    AVG(packets) AS avg_packets,
    SUM(packets) AS sum_packets,
    AVG(bytes) AS avg_bytes,
    SUM(bytes) AS sum_bytes
    FROM df
    WHERE dnsQName IS NOT NULL
    GROUP BY dnsQName
    ORDER BY sum_bytes DESC
    """
  ).count()
```

6. **Aggregate**: Group by dip, average packets and bytes, and sort by average bytes for a given start and end date.
   Aggregate also uses non-partitioned fields.



```
var input_df = (spark.read.
  fields(
    "sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes",
    "startTime", "endTime", "ipfix:observationDomainId", "ipfix:vlanId", "ipfix:silkAppLabel",
    "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName",
    "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address"
  )
  .ipfix(s3Repository=input_data, enableNewRelation=new_relation.toBoolean, targetSize=target_size.toLong)
)
val (aggTime, aggCount): (Long,Long) = time {
  var https_flows = input_df.filter(stime_range(start_date, end_date)).groupBy($"dip").avg("packets", "bytes").sort($"avg(bytes)".desc)
  https_flows.count()
```

4 Test Results

This section summarizes our test results. We used various tools, such as Spark-Bench, Ganglia, and Spark metrics sinks, to capture performance statistics during test operations. We focused on the following:

- decreasing the amount of time the analyst must wait for query results
- ensuring the system had enough resources to process the workload

We evaluated several system configurations over the last few months of testing and made changes to address issues as they arose and optimize performance.

We implemented several code changes to optimize the use of the AWS S3 service. These changes reduced the amount of time required to read data, thus reducing overall processing time.

4.1 Data Storage

We chose S3 as the storage solution for EMR. There are several benefits of using S3 over HDFS in the cloud. To take advantage of the elastic capabilities of an EMR cluster, S3 and EMRFS are requirements. With HDFS as the storage solution, persistent clusters must be used and are not able to scale-in to save on instance costs.

Core nodes must remain active to keep data in place, even when workloads are not running. This eliminates the flexibility of a cloud solution leading to higher costs than on-premises clusters. If a decision is made to store data in HDFS for performant workloads, due diligence must be taken to establish backup and restore procedures for the data.

Because of the virtual nature of cloud resources, the need to rebuild or restart clusters is not uncommon. If a master node is terminated, the whole cluster is terminated, which results in the loss of any persisted data. This is also the case for upgrading clusters with new versions of services like Spark or Hadoop.

S3 is an object store, not a directory-based file store like those on Linux operating systems. In particular, object stores lack things such as hierarchical files structuring, file locking, and file consistency. Instead, textual keys are mapped to binary values within a given object.

AWS calls its top-level grouping of objects “buckets.” When you create a cluster with consistent view enabled, Amazon EMR uses an Amazon DynamoDB database to store object metadata and track consistency with Amazon S3. If consistent view determines that Amazon S3 is inconsistent during a file system operation, it retries that operation according to rules you define. Using consistent view incurs DynamoDB charges, which are typically small, in addition to the charges for Amazon EMR.

Another benefit of using S3 over HDFS is that you can use S3 as a central repository for all your analytic workloads. Multiple EMR clusters, as well as other services such as Amazon Athena and

Sagemaker, can be pointed at the same data simultaneously. S3 offers a highly reliable, durable, and redundant storage solution without the need for 3x replication.

We determined early in our testing that the engineering and operational impacts related to using HDFS outweighed the potential benefits of a near-term deployment. A downside of S3 is that storage is decoupled from compute, but in our testing, the performance benefits of HDFS were not significant enough to outweigh the flexibility of transient, elastic, S3-backed clusters. However, HDFS may be an option to support a point solution for special use cases in the future.

4.2 AWS Optimizations

Spark is responsible for planning jobs and allocating resources for executing queries. It matches partition selection criteria—query filters such as time range, agency, and application label—to S3 keys representing the location of IPFIX data matching those criteria. Left alone, Spark scans all the available keys in the bucket. With the large number of files and keys created per day, the Spark driver is likely to scan keys with data outside the user's query data range, and if not bounded, it is also at risk of running out of memory and having the job fail.

The Mothra S3 data source code limits key scanning to the time range covered by the partition selection criteria. While this optimization led to improved query times due to more efficient key scanning, it is still not guaranteed to be bounded. Further enhancements to the Mothra S3 code allow it to scan keys more deliberately.

In particular, we added settings for specifying the number of Spark tasks to create (numSlices) and the maximum size of any given task (targetSize). These settings allow for tuning based on the available hardware and data access patterns. We implemented a two-pass approach to allow for determining the correct numSlices parameter for a given targetSize (described above). This approach keeps each task bounded while ensuring there are enough tasks to complete the job.

4.3 Ingest Processing

We performed the majority of the testing using 50 collector simulators. We did this because of cost considerations and the complexity in developing a more advanced seed file. Large instance types are required to support the processing requirements. The packer node metrics illustrated in Figure 5 through Figure 8 are samples of metrics for the Mothra Packer cluster from the previous three months of ingest testing.

Mothra Packer was running on a separate EMR cluster with three r5.4xlarge Master instances. The 50 collector simulators were sending data to two master nodes (25x2) via rwsender/rwreceiver. Each of the two Packer nodes was ingesting 1.3 TB of uncompressed IPFIX data per day. Mothra Packer was set to partition on year, month, day, observationDomainId, and silkApplabel. These partitioned files were then written to an S3 bucket in the AWS us-gov-east-1 region.

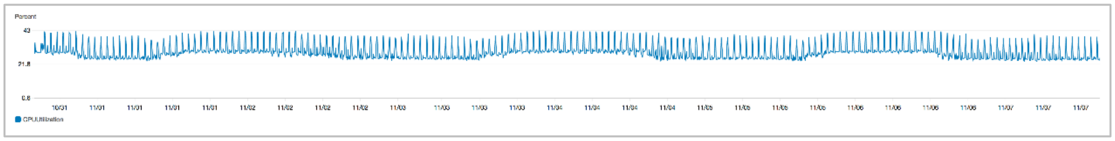


Figure 5: Packer Node 1 CPU Usage

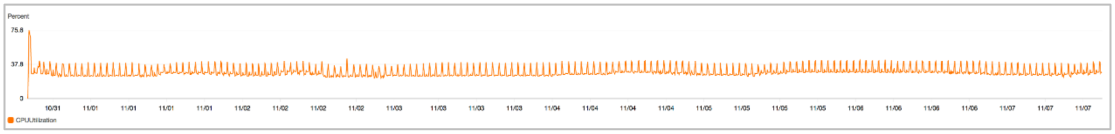


Figure 6: Packer Node 2 CPU Usage

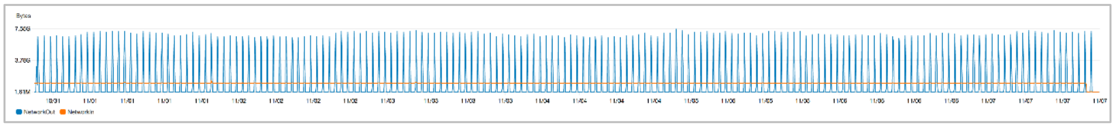


Figure 7: Packer Node 1 Network

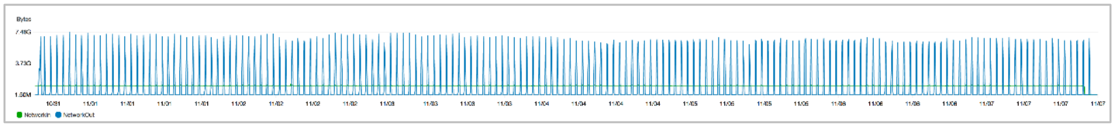


Figure 8: Packer Node Network

As we scaled up the data volume, processing limits were identified, resulting in a build-up of files waiting to be processed. We are performing additional work to isolate the root cause and determine the best path forward.

The design allows for horizontal and vertical scaling. Horizontally, we can add additional packer nodes to our cluster to help with ingest performance. Vertically, we can test larger master edge instances. This scaling relates to our theory that there is a correlation between the number of collectors and the number of cores available in the master node. We found that master nodes that had cores with (n-2) collectors connected were able to handle all traffic processing.

Another option is to tweak several variables to determine how to optimize the processing power of the master node's resources. This tweaking can be tested by changing variables one at a time and noting the impact on performance. Some of these variables include compression, Garbage Collection settings, polling intervals, # of threads (max pack jobs), and rotate delay.

Lastly, we began analyzing the time it takes Mothra Packer to process each incoming file and write it to disk. Currently, we are using EBS volumes for storage, which are not directly attached. Monitoring and collecting I/O metrics is helpful in determining throughput and latency.

4.4 Query Operations

The bulk of our effort was focused on measuring and reducing query time for analysts. It is important to understand that the testing was very iterative. The configuration of individual components and process settings has a significant impact on the functioning of the system. As test results were produced and analyzed, changes were made to the system to produce better performance. This iterative process allowed us to develop a better understanding of how the services are operating and which customizations are needed for this use case.

In the current configuration, searching a repository of more than 500 billion flows enables analysts to query a specific day and silkAppLabel (e.g., DNS or HTTPS) and receive a response in under five minutes. If an analyst is performing exploratory analysis, they can cache results so that subsequent queries access the data in memory instead on disk.

This feature should be used with caution; it should not be abused so that the system becomes memory constrained. As mentioned in Section 2.1, derived data sets provide an initial, fast capability for querying based on a specific indicator. In this case, Mothra is pre-populating results based on known fields of interest. The times in Table 4 do not reflect the situations when analysts use this method for the initial investigation.

4.4.1 Automated Testing Using Spark-Bench

We performed automated testing using Spark-Bench, enabling us to test various configurations and data sets. We ran tests with all six query types on the IPFIX-repeater repository stored in S3. At the time the queries were run, this repository included more than 500 billion flows. We ran tests using Mothra to filter time ranges that included 5.76 billion flows and 40.4 billion flows.

The flow repository we used for these automated tests originally did not contain any records with a silkAppLabel equal to zero. Given the partitioning scheme used for Mothra, grouping records by silkAppLabel at the last level, all of these extra flows would end up in their own partition. That means that any query that uses a filter of silkAppLabel not equal to zero would perform similarly, even with significant silkApplabel equal to zero flows present.

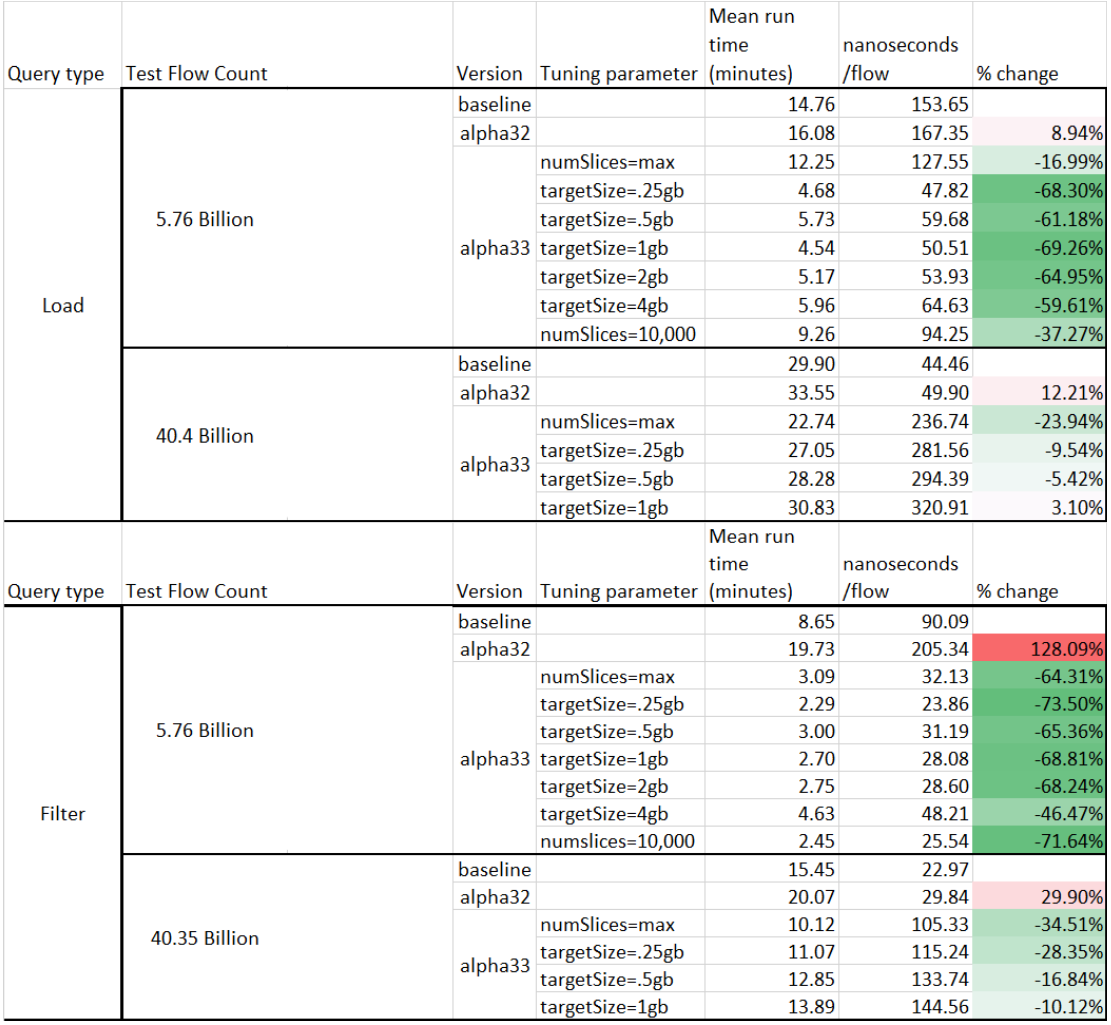
While there would be additional minor overhead of ruling out the partitions with files containing records with silkAppLabel equal to zero if they were present, we are confident that the test queries would perform comparably on repositories with five times the number of flows if they had significant flows with silkAppLabel equal to zero.

Three versions of Mothra were tested:

- Mothra 1.3.2 release was used as a baseline.
- Mothra 1.3.2-alpha-32 included some driver memory management.
- Mothra 1.3.2 alpha-33 included more efficient S3 object access and tunable driver memory management.

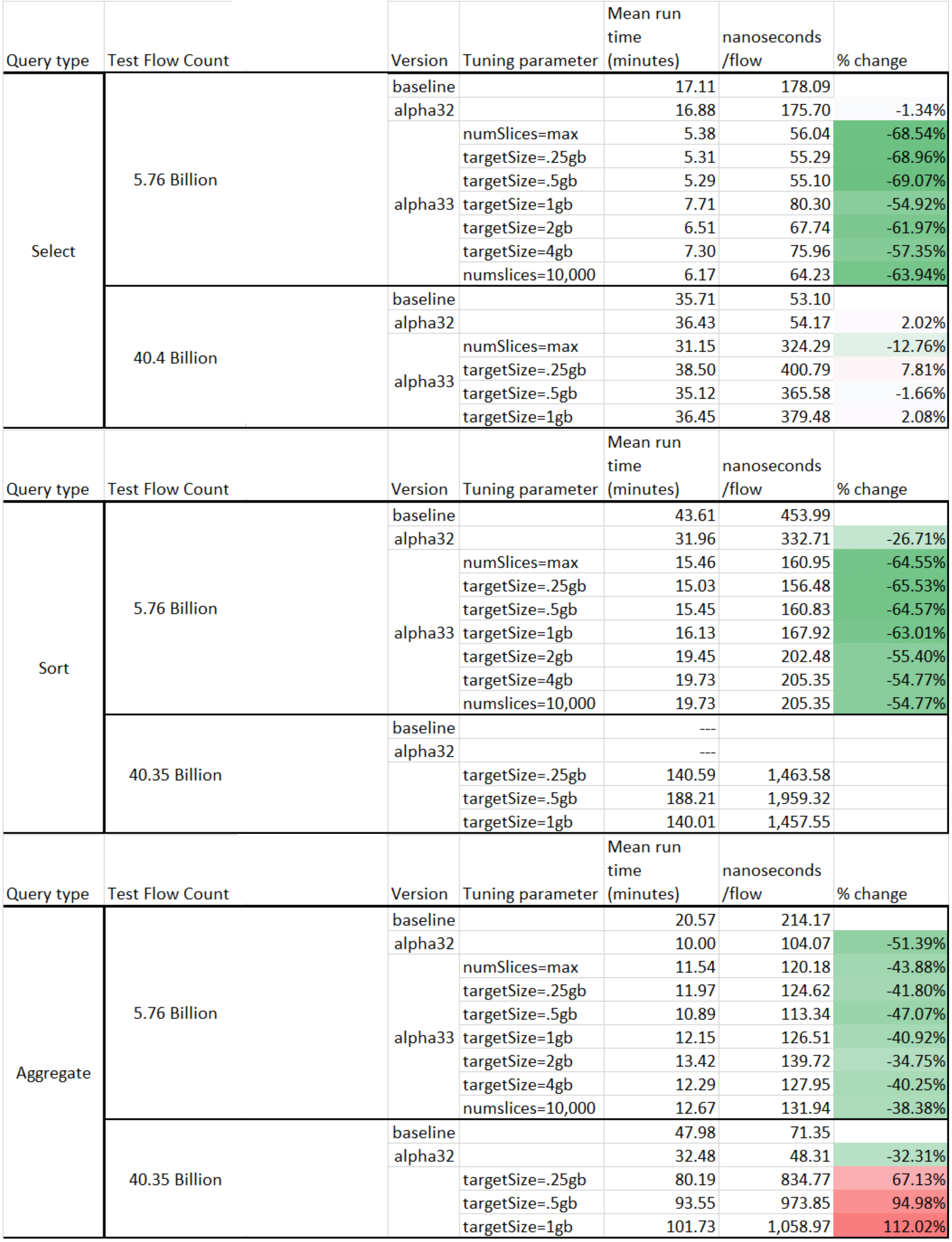
With Mothra 1.3.2-alpha-33, numSlices and targetSize variables were included for driver memory optimization and query performance. Table 4 illustrates the results of multiple iterations of each test and the percent change in run time as compared to the baseline test.

Table 4: Automated Testing Results



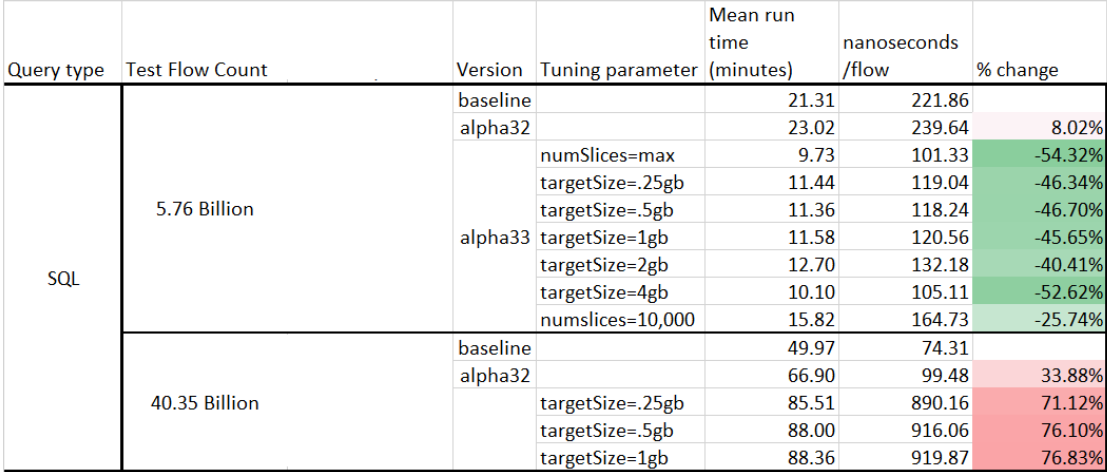
Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
Load	5.76 Billion	baseline		14.76	153.65	
		alpha32		16.08	167.35	8.94%
		alpha33	numSlices=max	12.25	127.55	-16.99%
			targetSize=.25gb	4.68	47.82	-68.30%
			targetSize=.5gb	5.73	59.68	-61.18%
			targetSize=1gb	4.54	50.51	-69.26%
			targetSize=2gb	5.17	53.93	-64.95%
			targetSize=4gb	5.96	64.63	-59.61%
			numSlices=10,000	9.26	94.25	-37.27%
	40.4 Billion	baseline		29.90	44.46	
		alpha32		33.55	49.90	12.21%
		alpha33	numSlices=max	22.74	236.74	-23.94%
			targetSize=.25gb	27.05	281.56	-9.54%
			targetSize=.5gb	28.28	294.39	-5.42%
			targetSize=1gb	30.83	320.91	3.10%
Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
Filter	5.76 Billion	baseline		8.65	90.09	
		alpha32		19.73	205.34	128.09%
		alpha33	numSlices=max	3.09	32.13	-64.31%
			targetSize=.25gb	2.29	23.86	-73.50%
			targetSize=.5gb	3.00	31.19	-65.36%
			targetSize=1gb	2.70	28.08	-68.81%
			targetSize=2gb	2.75	28.60	-68.24%
			targetSize=4gb	4.63	48.21	-46.47%
			numslices=10,000	2.45	25.54	-71.64%
	40.35 Billion	baseline		15.45	22.97	
		alpha32		20.07	29.84	29.90%
		alpha33	numSlices=max	10.12	105.33	-34.51%
			targetSize=.25gb	11.07	115.24	-28.35%
			targetSize=.5gb	12.85	133.74	-16.84%
			targetSize=1gb	13.89	144.56	-10.12%

Table 4: Automated Testing Results (continued)



Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
Select	5.76 Billion	baseline		17.11	178.09	
		alpha32		16.88	175.70	-1.34%
		alpha33	numSlices=max	5.38	56.04	-68.54%
			targetSize=.25gb	5.31	55.29	-68.96%
			targetSize=.5gb	5.29	55.10	-69.07%
			targetSize=1gb	7.71	80.30	-54.92%
			targetSize=2gb	6.51	67.74	-61.97%
			targetSize=4gb	7.30	75.96	-57.35%
			numslices=10,000	6.17	64.23	-63.94%
	40.4 Billion	baseline		35.71	53.10	
		alpha32		36.43	54.17	2.02%
		alpha33	numSlices=max	31.15	324.29	-12.76%
			targetSize=.25gb	38.50	400.79	7.81%
			targetSize=.5gb	35.12	365.58	-1.66%
			targetSize=1gb	36.45	379.48	2.08%
Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
Sort	5.76 Billion	baseline		43.61	453.99	
		alpha32		31.96	332.71	-26.71%
		alpha33	numSlices=max	15.46	160.95	-64.55%
			targetSize=.25gb	15.03	156.48	-65.53%
			targetSize=.5gb	15.45	160.83	-64.57%
			targetSize=1gb	16.13	167.92	-63.01%
			targetSize=2gb	19.45	202.48	-55.40%
			targetSize=4gb	19.73	205.35	-54.77%
			numslices=10,000	19.73	205.35	-54.77%
	40.35 Billion	baseline		---		
		alpha32		---		
			targetSize=.25gb	140.59	1,463.58	
			targetSize=.5gb	188.21	1,959.32	
			targetSize=1gb	140.01	1,457.55	
Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
Aggregate	5.76 Billion	baseline		20.57	214.17	
		alpha32		10.00	104.07	-51.39%
		alpha33	numSlices=max	11.54	120.18	-43.88%
			targetSize=.25gb	11.97	124.62	-41.80%
			targetSize=.5gb	10.89	113.34	-47.07%
			targetSize=1gb	12.15	126.51	-40.92%
			targetSize=2gb	13.42	139.72	-34.75%
			targetSize=4gb	12.29	127.95	-40.25%
			numslices=10,000	12.67	131.94	-38.38%
	40.35 Billion	baseline		47.98	71.35	
		alpha32		32.48	48.31	-32.31%
			targetSize=.25gb	80.19	834.77	67.13%
			targetSize=.5gb	93.55	973.85	94.98%
			targetSize=1gb	101.73	1,058.97	112.02%

Table 4: Automated Testing Results (continued)

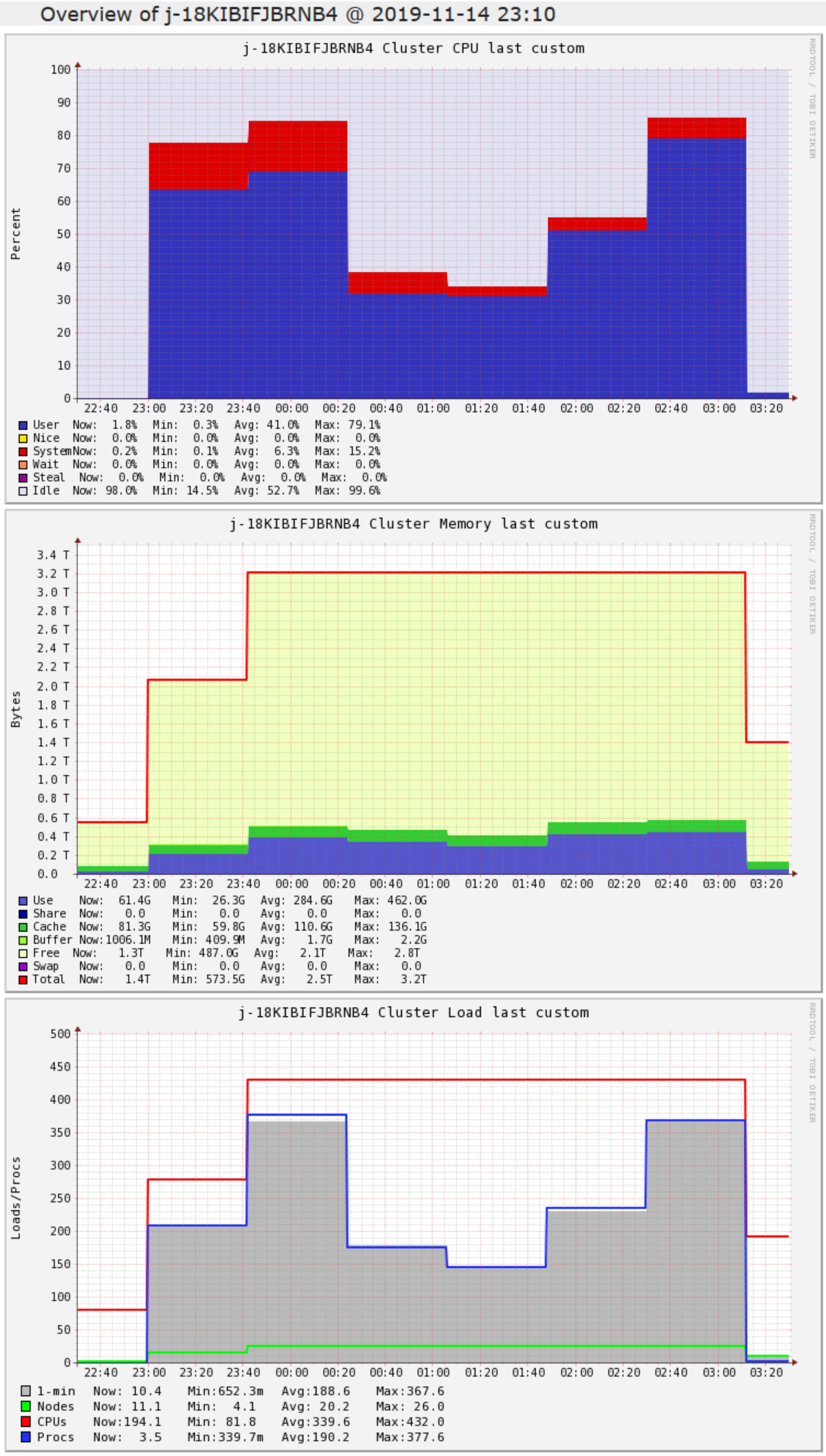


Query type	Test Flow Count	Version	Tuning parameter	Mean run time (minutes)	nanoseconds /flow	% change
SQL	5.76 Billion	baseline		21.31	221.86	
		alpha32		23.02	239.64	8.02%
		alpha33	numSlices=max	9.73	101.33	-54.32%
			targetSize=.25gb	11.44	119.04	-46.34%
			targetSize=.5gb	11.36	118.24	-46.70%
			targetSize=1gb	11.58	120.56	-45.65%
			targetSize=2gb	12.70	132.18	-40.41%
			targetSize=4gb	10.10	105.11	-52.62%
			numslices=10,000	15.82	164.73	-25.74%
	40.35 Billion	baseline		49.97	74.31	
		alpha32		66.90	99.48	33.88%
			targetSize=.25gb	85.51	890.16	71.12%
			targetSize=.5gb	88.00	916.06	76.10%
			targetSize=1gb	88.36	919.87	76.83%

More than 90 additional metrics were collected using two Spark metrics sinks:

- One writing Spark metrics to Ganglia. (See Figure 9.)
- One writing Spark driver memory metrics in one-second intervals to .csv files. (See Figure 10.)

We used these metrics to inform the development of new Mothra versions and to tune Spark configuration.



Overview of j-18KIBIFJBRNB4 @ 2019-11-14 23:10

j-18KIBIFJBRNB4 Cluster CPU last custom

```
■ User   Now:  1.8%   Min:  0.3%   Avg: 41.0%   Max: 79.1%
■ Nice   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
■ SystemNow:  0.2%   Min:  0.1%   Avg:  6.3%   Max: 15.2%
■ Wait   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
■ Steal   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
□ Idle   Now: 98.0%   Min: 14.5%   Avg: 52.7%   Max: 99.6%
```

j-18KIBIFJBRNB4 Cluster Memory last custom

```
■ Use     Now:  61.4G   Min:  26.3G   Avg: 284.6G   Max: 462.0G
■ Share   Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
■ Cache   Now:  81.3G   Min:  59.8G   Avg: 110.6G   Max: 136.1G
■ Buffer Now:1006.1M   Min: 409.9M   Avg:   1.7G   Max:   2.2G
□ Free   Now:   1.3T   Min: 487.0G   Avg:   2.1T   Max:   2.8T
■ Swap    Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
■ Total   Now:   1.4T   Min: 573.5G   Avg:   2.5T   Max:   3.2T
```

j-18KIBIFJBRNB4 Cluster Load last custom

```
■ 1-min   Now: 10.4   Min:652.3m   Avg:188.6   Max:367.6
■ Nodes   Now: 11.1   Min:  4.1    Avg: 20.2   Max: 26.0
■ CPUs    Now:194.1   Min: 81.8    Avg:339.6   Max:432.0
■ Procs   Now:  3.5   Min:339.7m   Avg:190.2   Max:377.6
```

Figure 9: Sample Ganglia Metrics (test run 11/11 22:30 – 11/12 03:30)

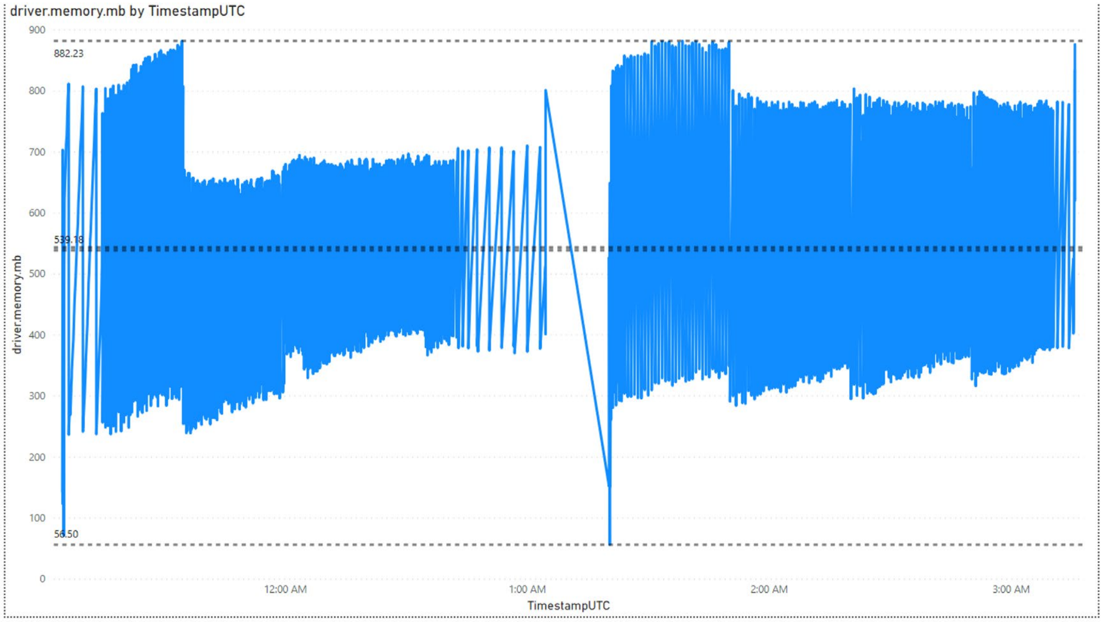


Figure 10: Sample Driver Memory Spark Metrics (test run 11/11 22:30 – 11/12 03:30)

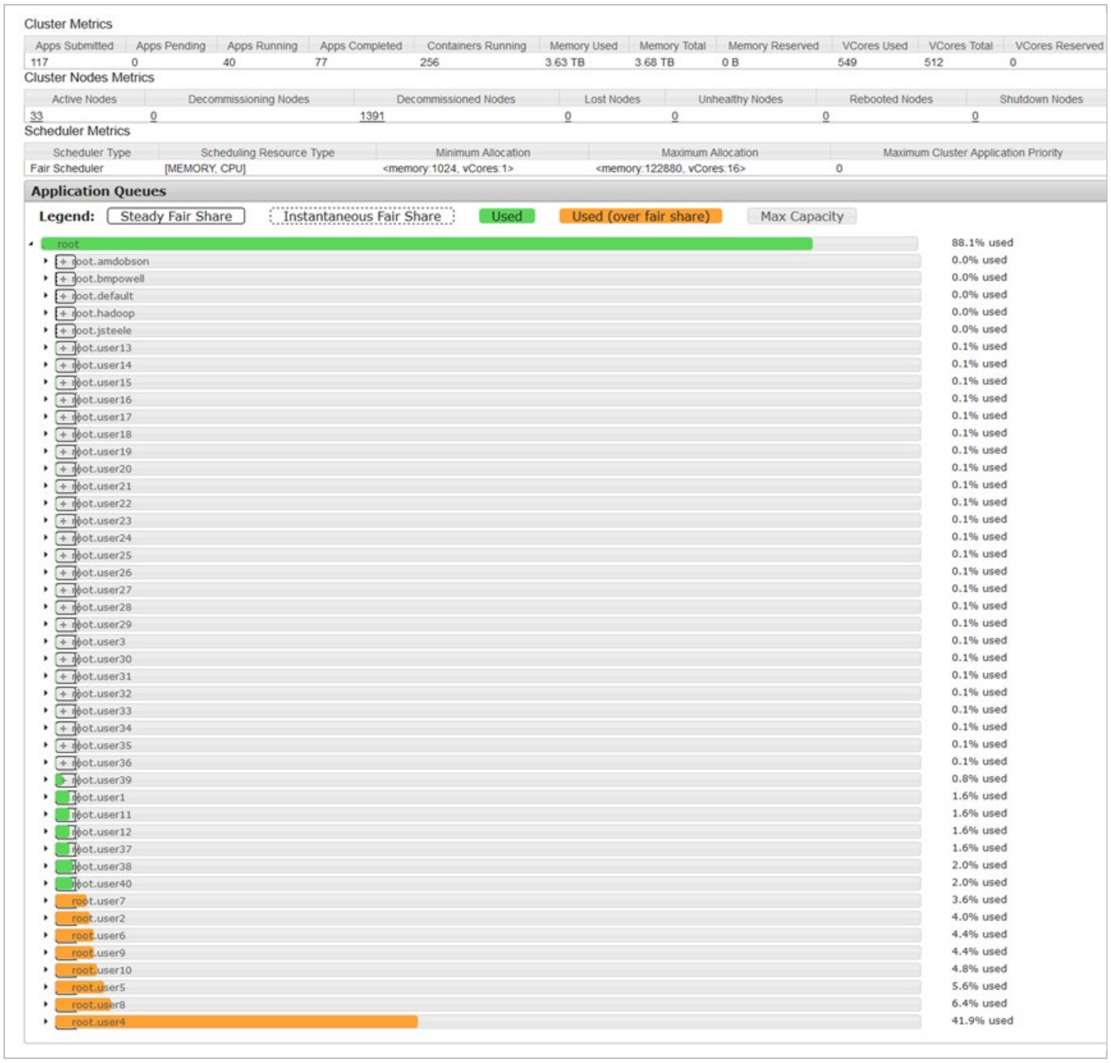
4.4.2 Manual Testing Using Jupyter

We ran tests that simulated 40 users running queries concurrently with Jupyter notebooks. We conducted these tests to confirm that there was not a 32-process limit as experienced in Zeppelin. We provided each user with a date range to find files and counted the number of flows within that range. We also made changes to each user's date range to apply some variance. The date ranges were as small as one day and as large as one month. Each notebook ran a filter, count, and show command.

Below are the steps we took to create the multi-user test:

1. Create a text file containing 40 test usernames, user[1-40].
2. Use Bash script to add users to JupyterHub via PAM authentication.
3. Use Bash script to create HDFS home directories for each test user.
4. Run a shared notebook on each user with cluster scaled to max resources available.
5. Monitor the Resource Manager to ensure that each user was allotted a sufficient amount of resources.

Each user received a percentage of the configured queue based on the resource demands of the query. Preemption settings allowed for more resources to be available for more expensive queries. As you can see in Figure 11, based on the variable queries, some jobs are just starting while others are requesting more resources based on the calculated size of the query.


Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
117	0	40	77	256	3.63 TB	3.68 TB	0 B	549	512	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
33	0	1391	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Fair Scheduler	[MEMORY, CPU]	<memory:1024, vCores:1>	<memory:122880, vCores:16>	0

Application Queues

Legend: Steady Fair Share | Instantaneous Fair Share | Used | Used (over fair share) | Max Capacity

Queue	Used
root	88.1% used
root.amdobson	0.0% used
root.bmpowell	0.0% used
root.default	0.0% used
root.hadoop	0.0% used
root.jsteele	0.0% used
root.user13	0.1% used
root.user14	0.1% used
root.user15	0.1% used
root.user16	0.1% used
root.user17	0.1% used
root.user18	0.1% used
root.user19	0.1% used
root.user20	0.1% used
root.user21	0.1% used
root.user22	0.1% used
root.user23	0.1% used
root.user24	0.1% used
root.user25	0.1% used
root.user26	0.1% used
root.user27	0.1% used
root.user28	0.1% used
root.user29	0.1% used
root.user3	0.1% used
root.user30	0.1% used
root.user31	0.1% used
root.user32	0.1% used
root.user33	0.1% used
root.user34	0.1% used
root.user35	0.1% used
root.user36	0.1% used
root.user39	0.8% used
root.user1	1.6% used
root.user11	1.6% used
root.user12	1.6% used
root.user37	1.6% used
root.user38	2.0% used
root.user40	2.0% used
root.user7	3.6% used
root.user2	4.0% used
root.user6	4.4% used
root.user9	4.4% used
root.user10	4.8% used
root.user5	5.6% used
root.user8	6.4% used
root.user4	41.9% used

Figure 11: Multi-User Test Monitoring

4.5 Elasticity and AWS Region Capacity

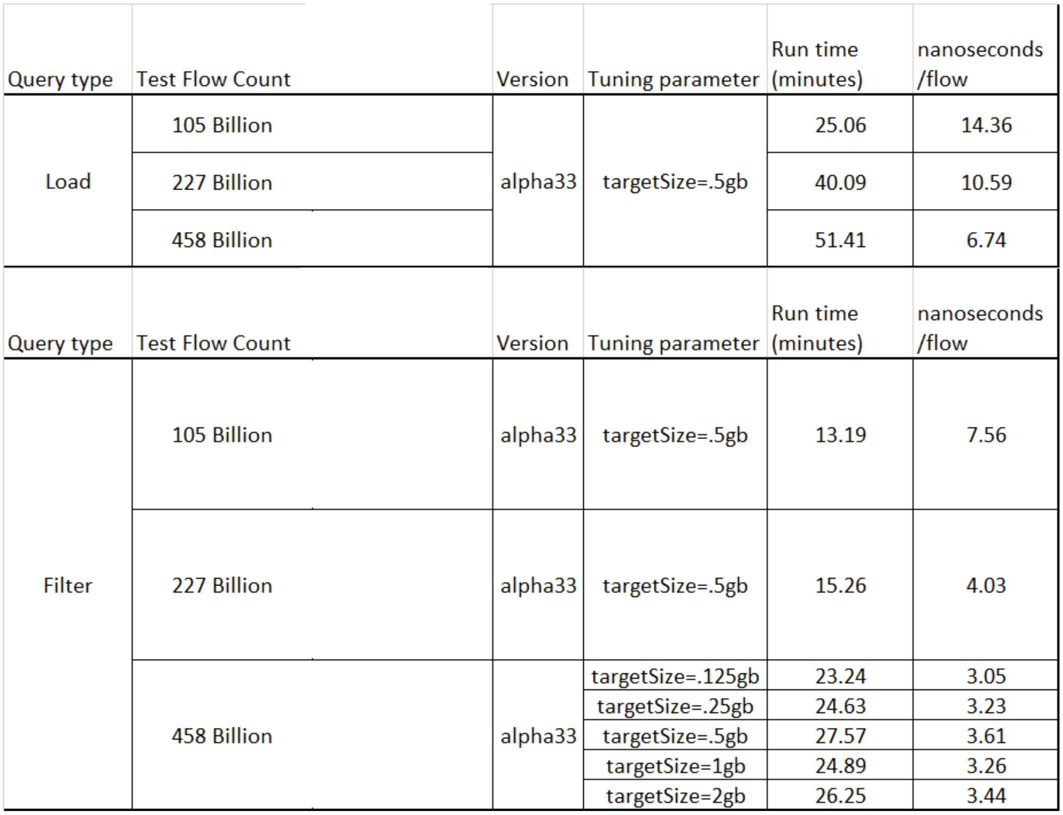
As we already mentioned in Section 2.2 (Prototyped Solution) and Section 2.4 (AWS Environment), we recommend elastic, auto-scaling clusters for query analysis. Elasticity provides many cost and performance benefits and is one of the main drivers for moving to the cloud.

During our testing in the us-gov-east-1 region, we found that we were reaching resource capacity limits (i.e., the types of instances that were chosen for the solution were not always available in the region selected). For example, our 40-node elastic EMRFS cluster would scale consistently only to about 27 nodes on average. After discussing our use case and these limits with an AWS EMR specialist, we learned that more capacity is available in the us-gov-west-1 region. There are other alternatives for securing more capacity as well, such as using reserved instances to guarantee capacity.

Using infrastructure-as-code and DevOps best practices, we were able to stand up a new 200-node elastic EMRFS cluster in the us-gov-west-1 region in **less than one business day**. This cluster included the configuration of Mothra, Spark-Bench, JupyterHub, and all other necessary services. We were also able to move over 200 TB of data (3.7 million objects) to the S3 us-gov-west-1 region in **two days**.

During initial testing, the cluster scaled out to as many as 167 r5.4xlarge nodes. This added capacity enabled us to run much larger queries in a short amount of time. Table 5 shows the results from the Load and Filter queries with the same repository containing over 500 billion flows using a Mothra date partition to filter a number of records before each operation. Notice that the load operation is reading all flows and performing a count. The filter operation includes the reading from disk times and is significantly shorter due to the partitioning scheme.

Table 5: Automated Testing Results (us-gov-west-1 region)



Query type	Test Flow Count	Version	Tuning parameter	Run time (minutes)	nanoseconds /flow
Load	105 Billion	alpha33	targetSize=.5gb	25.06	14.36
	227 Billion			40.09	10.59
	458 Billion			51.41	6.74

Query type	Test Flow Count	Version	Tuning parameter	Run time (minutes)	nanoseconds /flow
Filter	105 Billion	alpha33	targetSize=.5gb	13.19	7.56
	227 Billion	alpha33	targetSize=.5gb	15.26	4.03
	458 Billion	alpha33	targetSize=.125gb	23.24	3.05
			targetSize=.25gb	24.63	3.23
			targetSize=.5gb	27.57	3.61
			targetSize=1gb	24.89	3.26
			targetSize=2gb	26.25	3.44

4.6 Example Analyst Scenarios

The query operations we described so far were selected to evaluate the performance of the overall system, not mimic analyst scenarios. The situations described in this section demonstrate how the system responds to interactive analyst queries solving specific problems. The queries were run on a repository of synthetic data with more than 500 billion records, with “malicious” traffic occurring in short bursts on two different days. Any mention of “internal host” IPs, or “approved resolvers” are referencing **simulated** IP addresses labeled to reflect a more realistic and applicable scenario.

4.6.1 Response to {IP, destination port} Indicator

An analyst might be asked to look for traffic going to a specific IP/port combination and investigate any traffic they discover. The analyst first queries the DIP cache (not measured here due its already-established performance) and confirms that there was traffic sent from an IP to the malicious IP on a particular day. The analyst then queries Mothra to verify that the malicious destination port was used and retrieve all the flows for further analysis. The query for traffic from the source to the malicious IP, on the malicious port, for that specific day took **7.5 minutes** to complete.

The next step an analyst might take is to evaluate the discovered traffic is to see if the specific port is one that the potentially infected internal host IP commonly uses. A query for the previous week of data from the specific source IP being analyzed, coupled with a group of destination ports used and the record count for each, took **47 minutes** to complete. The query completed in **28 minutes** on a 100 node us-gov-west-1 elastic cluster.

4.6.2 Query for DNS Resolvers Used on a Particular Day

Most networks have security policies that specify which DNS resolvers are permitted to be used for internal hosts. Any DNS requests that use an unapproved resolver may indicate a misconfiguration or malicious activity. Analysts can automate daily queries to retrieve all resolvers used the previous day to investigate anomalies and build situational awareness of how the network is being used.

A query for all traffic from source IP addresses (silkAppLabel equal to 53) and eliciting a list of destination IP addresses took under **two minutes** to complete. If a destination port is used to identify DNS traffic instead of silkAppLabel, the response time increases to **14 minutes**. This time difference highlights the value of filtering using partitioning fields.

4.6.3 Query for Unapproved DNS Resolvers and Discovery of Changes

Once an approved list of DNS resolvers is established, an analyst can query to find new resolvers being used on the network. A query from source IPs (with silkAppLabel equal to 53) to destination IPs that are not in the “approved resolver” list took **under three minutes** to complete.

When an analyst finds an unapproved DNS resolver being used, it is informative to determine whether the IP address using this new resolver used an approved resolver in the past. A query to determine which DNS resolvers that the offending IP address used in the previous two weeks took **under two minutes** to complete.

5 Recommendations

Migrating Mothra to AWS provides significant improvements to performance and flexibility for future enhancements. We recommend the following actions for those migrating Mothra to AWS GovCloud to achieve results similar to the ones we experienced in our evaluation:

1. Incorporate new code enhancements, EMR, and Jupyter into the solution.
2. Deploy multiple EMR clusters to separate ingest and query processing.
3. Configure an elastic, auto-scaling cluster with preemption queues for query processing.
4. Deploy Elasticsearch/Kibana to support the cache capability.
5. Invest in DevOps practices and system monitoring/logging to support sustainment and operations responsibilities.
6. Consider archiving authoritative data from the sensors in a CISA data center for potential future uses if there is the possibility of using multiple cloud vendors over time.

Appendix Lessons Learned

As a result of our evaluation, we learned the following:

1. The AWS community and documentation are geared toward AWS commercial use. Rely on AWS support contacts for GovCloud specifics.

2. In AWS, the GovCloud west region is prioritized over the east region. Available capacity is limited in both regions, but especially in the east region. We were not able to fully scale clusters to certain node counts in the east region.

3. Auto-scaling is not always straightforward and sometimes fails without error messages due to capacity limits.

4. Long-running clusters must be destroyed to update EMR or service versions. Notebooks should be stored or backed up in S3 to avoid loss. Open-file limits might need to be tweaked if running large queries that reference a large number of files.

5. Implement a proper garbage collector to clear memory effectively and avoid out-of-memory errors in certain cases.

6. Best practices for recommended spark.executor.cores=5 are not ideal for the network flow use case since (1) many files exist and (2) there is a need to tune cores to be lower for optimized I/O.

7. Spark tuning requires extensive trial and error, especially with non-standard use cases or workloads.

8. Turn on monitoring of EMR clusters (cloud watch/ganglia) to verify expected behavior. For example, we found AWS bugs that prevent auto-scaling.

9. Use Yarn FAIR scheduler to ensure that users get a minimum share of cluster resources and allow for the preemption of resources.

10. Use FAIR scheduler to create queues “on the fly” for each user submitting jobs to a cluster.

11. With many users, good training and good citizenship is required. Use open notebooks to keep some resources and a Spark session even if no code is being executed.

12. Set configurations to release idle resources and cached data after a set time.

13. In AWS, Livy (Spark REST API) is used instead of direct Spark submit. Implement user impersonation to keep track of which users submit jobs. LDAP and PAM authentication are supported.

14. HDFS cluster testing did not demonstrate significantly increased performance.

15. Data stored in S3 enables elasticity and allows for access from multiple clusters and AWS services.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY**
(Leave Blank)

2. **REPORT DATE**
July 2021

3. **REPORT TYPE AND DATES COVERED**
Final

4. **TITLE AND SUBTITLE**
Experiences with Deploying Mothra in Amazon Web Services (AWS)

5. **FUNDING NUMBERS**
FA8702-15-D-0002

6. **AUTHOR(S)**
Brad Powell, Dan Ruef, & John Stogoski

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-2021-TR-007

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
SEI Administrative Agent
AFLCMC/AZS
5 Eglin Street
Hanscom AFB, MA 01731-2100

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
n/a

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

The Mothra large-scale data processing platform can be deployed in the AWS GovCloud environment. The Software Engineering Institute (SEI) evaluation of this deployment shows that it meets (and even exceeds) the operating requirements of the on-premises Mothra deployment. This report describes (1) how an SEI team developed an at-scale prototype of the on-premises system to test the performance of Mothra in the cloud and (2) the approaches the team recommends for similar deployments.

14. **SUBJECT TERMS**
large-scale data processing, AWS GovCloud, on-premises systems, cloud computing

15. **NUMBER OF PAGES**
36

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**
Unclassified

18. **SECURITY CLASSIFICATION OF THIS PAGE**
Unclassified

19. **SECURITY CLASSIFICATION OF ABSTRACT**
Unclassified

20. **LIMITATION OF ABSTRACT**
UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102