Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds

Shelan Perera*, Ashansa Perera*, Kamal Hakimzadeh
SCS - Software and Computer Systems Department
KTH - Royal Institute of Technology
Stockholm, Sweden.
Email: {shelanp,ashansa,mahh}@kth.se

Abstract—**Big data processing is a hot topic in today's computer science world. There is a significant demand for analysing big data to satisfy many requirements of many industries. Emergence of the Kappa architecture created a strong requirement for a highly capable and efficient data processing engine. Therefore data processing engines such as Apache Flink and Apache Spark emerged in open source world to fulfill that efficient and high performing data processing requirement.**

There are many available benchmarks to evaluate those two data processing engines. But complex deployment patterns and dependencies make those benchmarks very difficult to reproduce by our own.

This project has two main goals. They are making few of community accepted benchmarks easily reproducible on cloud and validate the performance claimed by those studies.

Keywords– **Data Processing, Apache Flink, Apache Spark, Batch processing, Stream processing, Reproducible experiments, Cloud**

I. INTRODUCTION

Today we are generating more data than ever. we are generating nearly 2.5 Quintillion bytes of data per day [1]. In a world of so much big data the requirement of powerful data processing engines is gaining more and more attention. During the past few years we could observe that there are many data processing engines that emerged.

Apache Spark [2] and Apache Flink [3] are such two main open source data processing engines. They were able to accumulate the interest of many data processing stakeholders due to their powerful architectures as well as higher level of performance claims.

There are many performance reports and articles about Apache Flink and Apache Spark published by both the industry and the academic institutes. But many studies are not reproducible conveniently even though there are many interested parties to reproduce those results by themselves. There are many valid reasons to reproduce the results because most of the original results are not generic enough to be adopted by others. The requirements may vary from changing the deployment pattern to fine tuning parameters for specific use cases. Data processing engines such as Apache Flink and Apache Spark have such many variables, which can be adjusted so reproducing a completed experiment with different settings is a frequent requirement.

Karamel [4] a deployment orchestration engine which is developed to design and automate reproducible experiments on cloud and bare-metal environments. It provides a convenient GUI to design reproducible experiments and also a Domain Specific Language (DSL) to declare dependencies and software tools that are required to setup and run the experiment. In this project we are using Karamel to make the distributed experiments reproducible conveniently. Further we are making the experiments and its resources available in Github to facilitate the reproducing those experiments conveniently.

A. Scope

In this project we are integrating batch processing and stream processing benchmarks to Karamel. We chose Apache Spark and Apache Flink mainly because of their reputation in the data processing world as well as the interest that is being shown by different communities to compare them against known benchmarks. In this report we are evaluating the performance of those two engines with Karamel and discuss how Karamel helps to achieve the goal of reproducibility conveniently.

---

* These authors contributed equally to this work.

II. RELATED WORK

We evaluated few benchmarks which are popular among the data processing communities. In this project two different benchmarks were used to design experiments for the batch processing and the stream processing.

A. Batch Processing

For batch processing we used Terasort [5] benchmark which was initially developed as a benchmark for evaluating Apache Hadoop Map reduce jobs. This benchmark has been used by many performance validations and competitions, hence considered as one of the most popular applications to benchmark batch processing applications. It was enhanced further to benchmark Apache Flink and Apache Spark by modifying the Map Reduce functionality.

We integrated the Terasort applications developed by Dongwong Kim [6] for Apache Flink and Apache Spark. He evaluated few data processing frameworks with great details in a clustered environment. His experimental analysis provided a great foundation to start our analysis. We could reuse his experimental code developed for Terasort as it was already analyzed and accepted as a fair analysis to compare the performance by the open source communities.

B. Stream Processing

For stream processing we researched two main stream processing benchmarks. They were Intel HiBench Streaming benchmark [7] and Yahoo Stream benchmark [8], which were recently developed to cater the requirement of comparing stream data processing. We evaluated Yahoo Stream benchmark as it includes a more realistic demo of a sample application which simulates an advertising campaign. But We Karamalized HiBench to make it reproducible via Karamel.

C. Karamel orchestration Engine

For deployment orchestration we used karamel and reused Chef cookbooks which were developed for clustered deployment of required software with some modifications to adopt them into Karamel. Those cookbooks are building blocks of Karamel, and Karamel orchestrates the deployment by managing dependencies. We published our cookbooks and Karamel Configurations in Karamel-lab [9] so a user can reuse those cookbook definitions in their experiments.

III. METHODOLOGY

This section describes the methodology we used in developing the reproducible experiments. In the first sub section the integration of the Karamel will be briefly described. Next subsections will describe both batch processing and stream processing experiments that were designed to reproduce them on a public cloud.

A. Using Karamel to design the experiments

Karamel is a web application that can be deployed in your local machine and access its UI through your browser. There are two main logical phases that an experiment designer has to follow when designing an experiment. These phases are logically separated, so parts of the experiments can be developed individually and integrated later into a one experiment. But there is no technical limitation to develop them as a monolithic experiment in one phase.

The first logical phase is the construction of the software-component-deployment and the second logical phase is to develop the experiment that should be run on the deployment.

The user needs to provide a configuration file which is written in a DSL of Karamel to orchestrate the deployment. The configuration file consists of Karamlized Chef cookbooks and recipes, deployment platform (such as EC2 or Google Cloud) and Node counts. Listing 1 shows a simple hadoop deployment with 3 servers on EC2. Line number 9 shows the cookbook which is used in deployment and their recipes are listed under $recipes$: sections, which begins with line number 16 and 23.

```
1  name: ApacheHadoopDeployment
2  #cloud provider
3  ec2:
4      type: m3.medium
5      region: eu-west-1
6  #cookbooks and recipies used in deployment
7  cookbooks:
8    hadoop:
9      github: "hopshadoop/apache-hadoop-chef"
10     branch: "master"
11
12 groups:
13   namenodes:
14 #No of Name nodes in the deployment
15     size: 1
16     recipes:
17         - hadoop::nn
18         - hadoop::rm
19         - hadoop::jhs
20   datanodes:
21 #No of Data nodes in the deployment
22     size: 2
23     recipes:
24         - hadoop::dn
25         - hadoop::nm
```

Listing 1. Sample configuration file for deploying a Hadoop cluster

This eliminates the burden of deployment and provides a very convenient way to deploy necessary clusters or software on the cloud with a single click.

The second logical phase can be accomplished with the assistance of inbuilt experiment designer [10] in Karamel. A user can use that experiment designer to develop an experiment or modify an existing experiment which is available in the Github. We designed our experiments using the skeletons developed by experiment designer. Listing 2. shows an extract from the generated cookbook which includes the core logic to run an experiment.

There are two ways that a user can design the execution of the experiment. One such way is, a user can combine both the deployment of clusters and execution of the benchmark in a single configuration file. When a user declares a cluster deployment as a dependency for benchmark execution, Karamel creates an intelligent execution graph considering (Execution DAG) for those dependencies.

Fig. 1. shows a similar Visual DAG generated inside the Karamel. This is a sample DAG generated for Listing 1. configuration file. Since there are 3 nodes in the system, (1 name node and 2 data nodes) those executions could happen in parallel. Further if they have dependencies they are managed intelligently by Karamel and executed in correct dependent order.
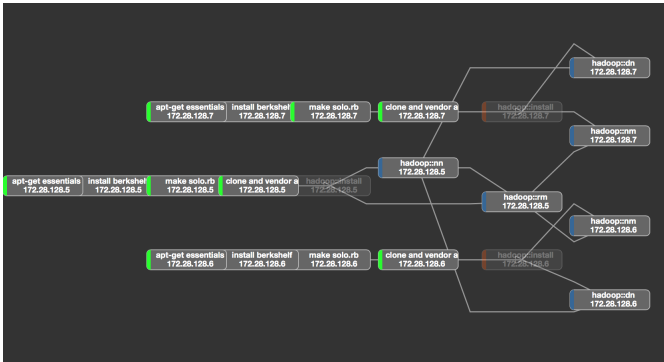

Fig. 1. Sample UI populated to change Apache Flink settings

B. Designing Batch processing Benchmark Experiment

Most of the benchmarks have two phases in the benchmark workflow. Benchmark needs to generate sample data to run the core application. Then application can execute its core logic with those input data.

1) Generating Input data for Terasort: Terasort produces input data using Teragen which internally uses Apache Hadoop map reduce jobs. This can generate large amounts of data that can be used by Terasort application. These generated data is stored in HDFS and used by both Apache Flink and Apache Spark applications.

One line of Teragen has 100 bytes of data. We generated 200GB, 400GB and 600GB of data in size using Teragen with Map Reduce jobs. Teragen data generation is developed as a separate experiment which will run before the experiment.

Listing 2. is the core logic which runs the Teragen application. There is an input parameter [:teragen][:records] (appears in line no 8.) , which can be used to configure the amount of data that needs to be generated. When we launch the experiment we can give the user input from the Karamel UI using generated input fields.

```
1 script 'run_experiment' do
2   cwd "/tmp"
3   user node['teragen']['user']
4   group node['teragen']['group']
5   interpreter "bash"
6   code <<-EOM
7 /srv/hadoop/bin/hadoop fs -rmr /srv
8 /srv/hadoop-2.4.0/bin/hadoop jar /srv/hadoop
    -2.4.0/share/hadoop/mapreduce/hadoop-
    mapreduce-examples-2.4.0.jar teragen #{
    node[:teragen][:records]} /srv/teragen
9   EOM
10 end
```

Listing 2. Part of the Teragen Experiment Cookbook

2) Running Terasort Experiment: We developed two experiments for Terasort using Karamel experiment designer similar to Teragen. Both these experiments were ran against the generated data of 200GB, 400GB and 600GB.

C. Designing Streaming Benchmark Experiment

Streaming benchmarks have a different workflow than the batch processing workflow. In streaming applications data needs to be generated continuously throughout the lifecycle of the experiment. Therefore our experiment was designed to have parallel execution flows for both data generation and data processing by benchmark. As previously stated we made both Yahoo Streaming benchmark and HiBench Streaming benchmark reproducible with Karamel.

1) Yahoo Streaming Benchmark: Yahoo Streaming Benchmark is designed to test Apache Flink, Apache Spark and Apache Storm [11] streaming performances. This requires additional deployments of Apache Zookeeper [12] and Apache Kafka[13] clusters for high throughput stream generation as input data.

This Benchmark simulates a real world advertisement campaign. It generates ad-campaign data using data generation scripts written in Clojure. All meta data related to ad-campaign is stored in Redis server with the relevant timestamps. Therefore this benchmark updates the Redis database periodically with application level metrics such as latency and throughput. It exposes two parameters to change the behavior of the test. They are:

- Time duration for Test
- No of records sent per second.

By changing those two parameters we could change the level of stress we are imposing at the system. We ran our reproducible benchmarks by changing those parameters and measured how Apache Spark and Apache Flink perform under those conditions.

We developed Cookbooks to automate the deployment and execution with Karamel. Therefore a user can reproduce the experiments we performed in their cloud environments.

2) HiBench Streaming Benchmark: Intel's Streaming benchmark is based on their HiBench benchmarking project [14]. They have developed it to test the performance of Apache Spark and Apache Storm. There are 7 micro benchmarks such as wordcount, grep and statistics etc. with different complexities and different workloads. Those benchmarks use pre-populated data based on a real world seed data set.

Since it did not have Apache Flink benchmark integrated, we needed to develop an equivalent Apache Flink micro benchmarks for the system. Their benchmark has a complex project structure as they have incorporated streambench as part of their HiBench suite which was developed initially for batch processing.

Similar to the Yahoo benchmark this also uses Apache Kafka and Apache Zookeeper clusters for high throughput stream data generation as the input. They offer different scales of data levels as well as different ways such as push or periodic data generation. In the push mode they try to generate as much as data possible. In the periodic data generation, the amount of data generated per second is throttled by the rate limit.

We also developed a cookbook to manage the lifecycle of the benchmark. This facilitated to execute the benchmark with Karamel as a reproducible experiment conveniently.

D. Monitoring Performance

In our experiments we collected two different types of performance metrics. They are application level per-formance metrics and system level performance metrics. For application level metrics we measured metrics such as execution time, latency and throughput. These measurements are experiment and application specific. Therefore we obtained these measurements relative to the experimental setup.

System level metrics can be application independent. Therefore we could use one system level measurement setup across different experiments. We evaluated different system level measurement tools and selected collectl [15] due to its light-weightiness and convenient deployment. We developed a comprehensive tool for managing the entire lifecycle of collectl, which we published as collectl-monitoring [16] in Github.

Using Collectl-monitoring we generated 4 reports for the system level measurements. They were CPU (CPU frequency and CPU load average per 5 Secs), Memory, Network and Disk utilization. This helps to measure the performance of the system in an application independent manner, without relying on the application specific measurements.

IV. EVALUATION

A. Batch Processing

Apache Spark and Apache Flink are memory intensive and process data in the memory without writing to the disks. Therefore we needed to plan our deployments to facilitate the requirements. In batch processing we also needed to store large amounts of data.

We performed few preparation experiments to estimate the storage overhead in our systems. In our experiments we could measure that 3 times of extra space is needed, compared to input data, to store the replicated data as well as temporary and intermediate results.

1) Configuration Details: We chose Amazon Elastic Computing Cloud (EC2) to run our evaluations and we had two types of nodes to be deployed. Some of the nodes act as Master nodes while other nodes act as slaves or worker nodes. Master nodes are processing a very low amount of data while worker nodes undertake the heavy data processing. Therefore, we differentiated them and chose two different types of EC2 instances. We chose m3.xlarge type instances for Master nodes while selecting i2.4xlarge for worker nodes.

Table 1. lists the configurations of i2.4xlarge instance and m3.xlarge instance. We used EC2 spot instances [17] as we can provide the spot request price limit in the Karamel configuration file, which became a cheaper

option for renting EC2 instances.

	Master (m3.xlarge)	Worker (i2.4xlarge)
CPU (GHz)	2.6	2.5
No of vCPUs	4	16
Memory (GB)	16	122
Storage :SSD (GB)	80	1600

Table I. Amazon EC2 configuration Details for Batch Processing Clusters

2) Deployment of Clusters: In our setup we had 3 main cluster deployments. They are Apache Hadoop cluster, Apache Flink cluster and Apache Spark cluster. Apache Hadoop cluster is used for running the map reduce programs for Teragen. Teragen produces input data which Apache Spark and Apache Flink clusters used for running the Terasort programs.

Table II. shows how the clusters are deployed in m3.xlarge machine and i2.4xlarge machine. We used 2 instances of i2.4xlarge and one instance of m3.xlarge.

Cluster	Master (m3.xlarge)	Worker (i2.4xlarge)	version
Hadoop	Name Node Resource Manager	Node Manager Data Node	2.4
Spark	Spark Master	Spark Worker	1.3
Flink	Job Manager	Task Manager	0.9

Table II. Cluster deployments for batch processing experiment

We constructed a Karamle definition and loaded into the Karamel engine to deploy all the clusters in a single execution.

3) Executing Terasort Experiment: After deploying the clusters using Karamel We ran Apache Spark's Terasort experiment and Apache Flink's Terasort experiments separately. After we launched the experiments, we started collectl-monitor to collect system level metrics for the running tests. Once the tests finished its execution, we stopped the collectl-monitor and generated the system level performance reports.

In our deployment we used default configurations and only changed the following parameters to maximize the utilization of memory.

- Apache Spark
  $spark.executor.memory = 100GB$
  $spark.driver.memory = 8GB$

- Apache Flink
  $jobmanager.heap.mb = 8GB$
  $taskmanager.heap.mb = 100GB$

These parameters were changed from Karamel configuration file. It is very convenient for a user to reproduce the results with different memory limits. Listing 3. shows how those parameters can be adjusted using Karamel Configuration file.

```
1 spark:
2    driver_memory: 8192m
3    executor_memory: 100g
4    user: ubuntu
5 flink:
6    jobmanager:
7      heap_mbs: '8192'
8    user: ubuntu
9    taskmanager:
10     heap_mbs: '102400'
```

Listing 3. Memory settings in Karamel Configuration file

Fig. 2 and Fig. 3 show how those memory parameters for Apache Spark and Apache Flink appear in Karamel configuration UI. This suggests the Karamel's convenience for different stakeholders with different requirements. Anyone who prefers automating everything can benefit from the comprehensive DSL based configuration file. Further, a user who wants to run the experiment adjusting few parameters can change those parameters through Karamel web application UI without knowing Karamel DSL and its configurations.
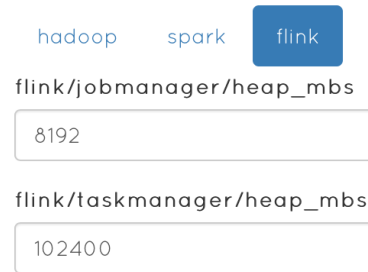

Fig. 2. Sample UI populated to change Apache Flink settings

Further, our experiments were designed to log the application level details and they were collected to draw the application level comparisons.After running Apache Spark's Terasort and Apache Flink's Terasort we plotted the execution times on a graph to compare the performances.

Fig. 4. compares the execution times logged at application level in the experiments. Those results were obtained at 3 different input levels of 200GB, 400GB and 600GB.
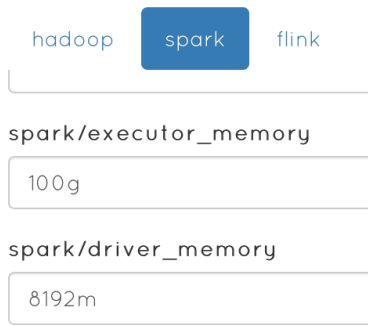
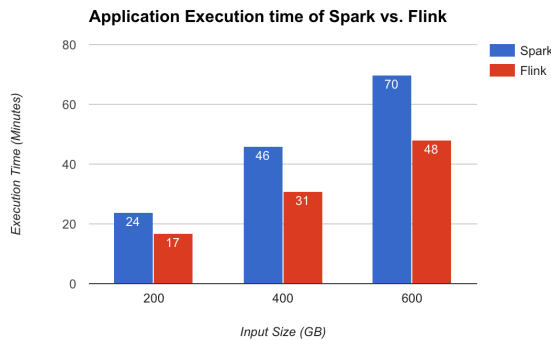Fig. 3. Sample UI populated to change Apache Spark settings

Fig. 4. Execution time comparison for different input sizes

We can observe that for all 3 input sizes Flink has recorded less execution time. Therefore Apache Flink has out performed Apache Spark in Terasort application for all 3 workloads we tested. On average Flink was 1.5 times faster than Spark for Terasort application.

Apache Flink is performing better than Apache Spark due to its pipelined execution. This facilitates Flink engine to execute different stages concurrently while overlapping some of them.

For the better explorations we also obtained system level metrics using collectl-monitor and the following graphs illustrates the impact of pipelined execution of Apache Flink.

Fig. 5 and Fig. 6 illustrates the CPU utilization of the two systems.

In Fig. 6 We can observe that Apache Spark is reaching almost 100 % during the execution. But Apache Flink executed the same load with less utilization as we can observe in Fig. 5. This is a consequence of the pipelined execution as Apache Flink stress the system smoothly while Spark is having significant resource utilization at certain stages.

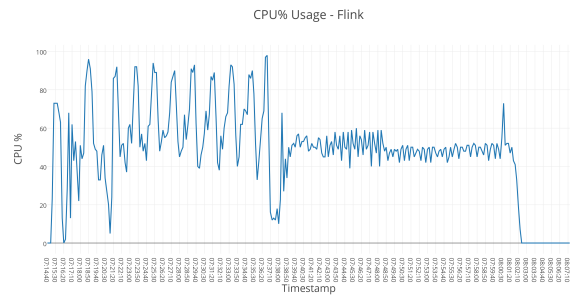Fig. 7 and Fig 8 illustrate the Memory utilization of

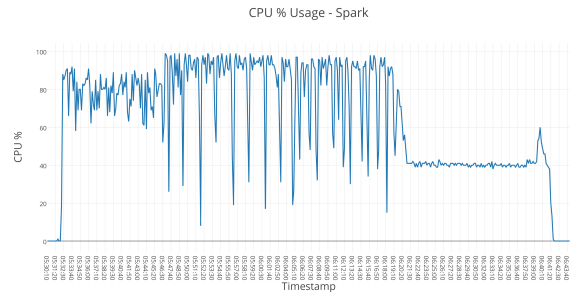Fig. 5. CPU utilization of Apache Flink in Batch processing

Fig. 6. CPU utilization of Apache Spark in Batch processing
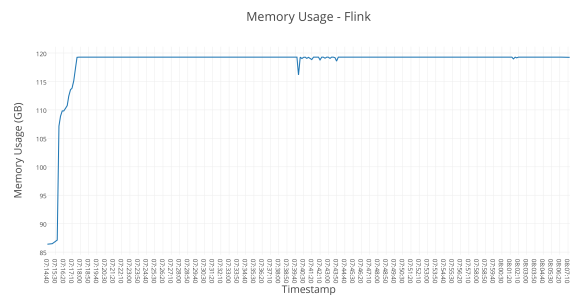
the two systems.

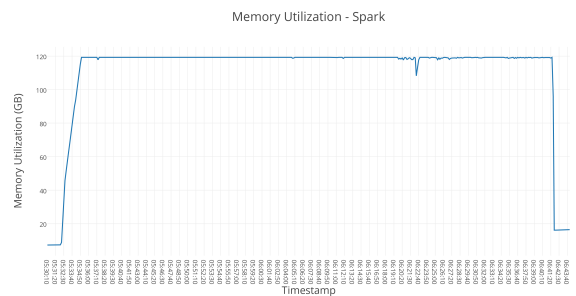Fig. 7. Memory utilization of Apache Flink in Batch processing

Fig. 8. Memory utilization of Apache Spark in Batch processing

In Fig. 7 and Fig 8 we can observe that both systems utilized all of the memory provided. This is because they

are trying to optimize their executions by processing as much as data in the memory without writing to the disks.

Fig. 9. and Fig. 10. illustrate the Disk utilization while Fig 11. and Fig 12. illustrate the Network utilization.
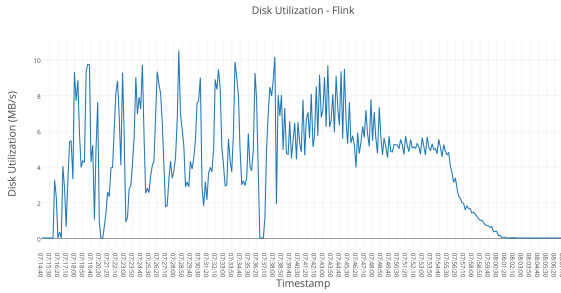

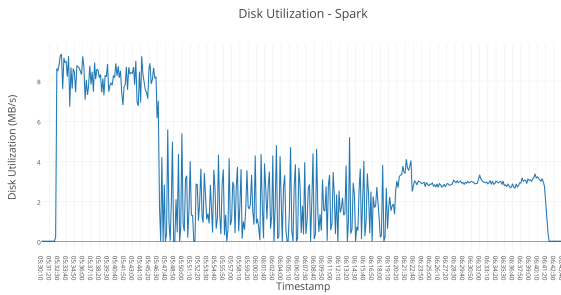Fig. 9. Disk utilization of Apache Flink in Batch processing

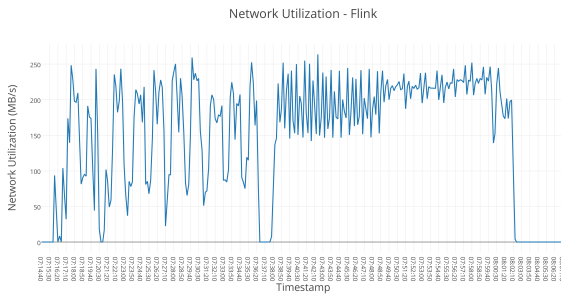Fig. 10. Disk utilization of Apache Spark in Batch processing

Fig. 11. Network utilization of Apache Flink in Batch processing

When we compare the disk utilization in the Fig. 9. and Fig. 10 , we can observe that Apache Spark clearly shows 3 different phases while Apache Flink shows a similar disk utilization behavior through out. Similarly When we compare the network utilization in Fig. 11 and Fig. 12, we can identify that Apache Spark is having a very insignificant network utilization during the first phase. Since Apache Flink has pipelined execution, it initiates network transfer from beginning.
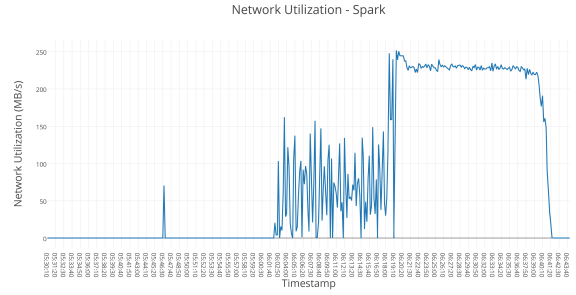

Fig. 12. Network utilization of Apache Spark in Batch processing

B. Stream Processing

In the stream processing we chose Yahoo Stream benchmarking suite to obtain the comparison results for Apache Flink and Apache Spark. We chose m3.xlarge instances for the low processing nodes similar to our batch processing experiments. But for the high processing node we selected r3.2xlarge instances, as they are optimized for memory intensive applications. i2 instances, which we used for batch processing provided both storage and memory optimizations. But for our streaming application there was a very less storage utilization.

Table III. shows the details of the configurations we used for streaming applications.

	Master (m3.xlarge)	Worker (r3.2xlarge)
CPU (GHz)	2.6	2.5
No of vCPUs	4	8
Memory (GB)	16	61
Storage :SSD (GB)	80	160

Table III. Amazon EC2 configuration Details for Streaming clusters

1) Deployment of Clusters: In our setup we had 3 main cluster deployments as similar to batch processing deployments. They were Apache Hadoop cluster, Apache Flink cluster and Apache Spark cluster. Additionally Apache Zookeeper and Apache Kafka were deployed for generating input data. We deployed them in the high processing node without making them fully distributed, to match the configurations of the Yahoo benchmark.This deployment facilitated minimal modifications to the benchmark execution scripts.

Table IV. shows how the clusters are deployed in m3.xlarge machine and r3.2xlarge machine.In this deployment we used 1 instance of r3.xlarge and one instance of m3.xlarge.

Cluster	Master (m3.xlarge)	Worker (i2.4xlarge)	version
Hadoop	Name Node	Node Manager	2.7.1
	Resource Manager	Data Node	
Spark	Spark Master	Spark Worker	1.5.1
Flink	Job Manager	Task Manager	0.10.1
Zookeeper	-	Zookeeper	3.3.6
Kafka	-	Kafka brokers	0.8.2.1
Redis	-	Redis database	3.0.5

Table IV. Deployment versions of clusters

Similar to the approach that we followed in the batch processing experiment we deployed the clusters using Karamel with a Karamel configuration file we developed. These scripts are hosted in Karamel lab [9] for a user to reproduce easily in a cloud environment.

We ran the benchmarks with the assistance from Karamel for different streaming rates. We configured event rates, starting from 1000 events per seconds up to 10000 events per second by increasing the rate with 1000 steps.

Fig. 13 shows the behavior of Apache Spark and Apache Flink with different streaming events rates. This graph was plotted by aggregating the latencies of different stream executions and obtaining 99th percentile of the data respectively.
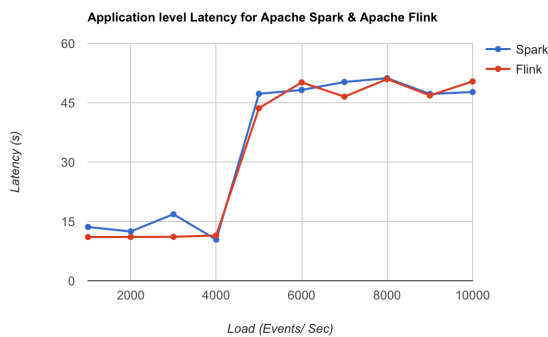

Fig. 13. Latency comparison for different event rates

We could observe that both Apache Flink and Apache Spark performed similarly under different loads. But we could observe that Apache Flink demonstrated a slightly better performance at lower event rates such as 1000, 2000 and 3000 events per second.

The reason for this slight improvement might have caused by Apache Flink's architecture. Apache Flink supports native stream execution but Apache Spark is approximating streaming by using a micro batch model. In Apache Spark they execute small micro batches per event window and apply operations.

There was a significant change in the latencies after 4000 events /sec input rate. This happened because of the Apache Kafka setup that was configured with the Yahoo Stream bench consisted only single broker for reducing the complexity. Therefore it could not handle that amount of input rate and generated a back pressure scenario.

In overall both Apache Spark and Apache Flink behaved in a similar manner for different input stream loads. In Yahoo benchmark the latencies reported, were the time between last event was emitted to kafka for a particular campaign window and when it was written into Redis. Therefore it is not the exact latency for Apache Flink or Apache Spark, but a relative measurement to compare the performance relatively.

We also collected system level metrics using Collectl-monitor. Fig. 14 and Fig. 15 shows the CPU utilization while running the benchmark. We could observe similar CPU behavior in both of the systems. Since this CPU utilization also includes the data initialization (seed data generation) and post processing (computing the latencies from Redis timestamps) of the Yahoo streaming benchmark, there are corresponding spikes for those stages in the system.

Fig. 16 and Fig. 17 has less memory utilization than the batch processing as we can see in Fig. 7 and Fig. 8. But in streaming Apache Flink consumed less amount of memory than Apache Spark. We can observe that Apache Flink consumed less than 14GB of memory while Apache Spark consumed around 18GB during its execution.
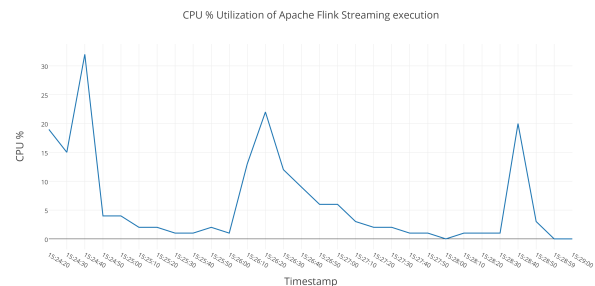

Fig. 14. CPU utilization of Apache Flink in Stream processing

Both application level and system level metrics do not clearly highlight, which data processing engine is performing better. Eventhough, we could observe that
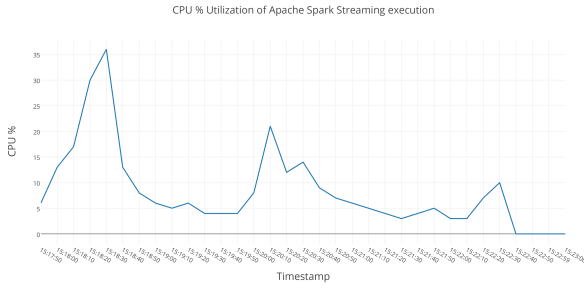
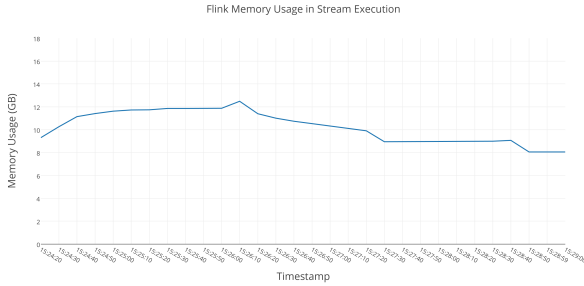Fig. 15. CPU utilization of Apache Spark in Stream processing

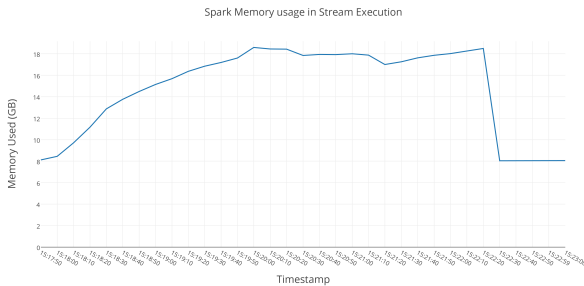Fig. 16. Memory utilization of Apache Flink in Stream processing

Fig. 17. Memory utilization of Apache Spark in Stream processing

Flink was performing slightly better in many occasions than Spark.

C. Reproducibility with Karamel

One of our main goals of the project is to make the benchmark applications conveniently reproducible. This convenient reproducibility covers two main phases. They are convenient deployment at deployment of the clusters, and convenient execution of the benchmark applications.

1) Convenient deployment of clusters: After starting the Karamel web application a user can reuse the Karamel configuration file for deploying the Apache Flink, Apache Hadoop, Apache Spark clusters. For stream processing applications we used two more deployments of Apache Kafka and Apache Zookeeper.

Both Karamel configurations are available at Karamel Lab [9] for a user to load into their own Karamel web application and deploy in their own cloud provider.

There is a flexibility for user to change the parameters through configuration file as mentioned in Listing 3. or to change them through the Karamel UI as shown in Fig 2. and Fig 3. Similarly no of nodes that needs to be deployed can be altered by changing the $size$ parameter (eg: listing 1. line no. 15 or line no. 22). So a user can recreate clusters very conveniently without undergoing the painful process of configuring the clusters.

2) Convenient execution of the benchmark: Similar to deploying the clusters a user can execute the benchmark on the deployed cluster. Similar to the cluster deployment, Parameters that are exposed can be changed either through UI parameter or using the Karamel configuration file. This allows a user to run the same experiment repeatedly many times with different settings.

Therefore, these features of Karamel provides a smooth end to end workflow for users to reproduce their experiments many times with much less overhead.

V. CONCLUSION

In this project we developed reproducible experiments for Apache Spark and Apache Flink in the cloud. Therefore another user who is interested in reproducing the same experiments can reproduce in their own cloud environments. Further they can easily change the exposed parameters to deeply analyse different behaviours at different scales.

We could achieve the goal of making the experiments publicly available by sharing all the reproducible artifacts online in Karamel Lab [9]. Further in our experiments we could validate the claims of the few studies such as Dongwong Kim's performance comparison which compared the performance of Apache Flink and Apache Spark. We could obtain similar characteristics and results in our setup therefore strengthening the results found by those initial comparisons.

VI. FUTURE WORK

Intel's HiBench and Yahoo Stream Benchmarks were developed recently. Therefore there are few issues that needs to be considered to evolve them Karamel. The main reason is they were designed to run manually in a deployed cluster not with an automated orchestrated environment like Karamel.

Yahoo's Benchmark experiment needs to be enhanced to run with a fully distributed clusters of Kafka and

Zookeeper to facilitate the experiment with very high input rates.

This complete project can be a basic framework for re-executing Apache Flink and Apache Spark benchmarks on public clouds to run different experiments in future.

REFERENCES

[1] "IBM - What is big data?" [accessed 6-January-2016]. [Online]. Available: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

[2] "Apache Spark™ - Lightning-Fast Cluster Computing," [accessed 6-January-2016]. [Online]. Available: https://spark.apache.org/

[3] "Apache Flink: Scalable Batch and Stream Data Processing," [accessed 6-January-2016]. [Online]. Available: https://flink.apache.org/

[4] "Karamel — Orchestrating Chef Solo," [accessed 6-January-2016]. [Online]. Available: http://www.karamel.io/

[5] "Hadoop TeraSort Benchmark - Highly Scalable Systems," [accessed 6-January-2016]. [Online]. Available: http://www.highlyscalablesystems.com/3235/hadoop-terasort-benchmark/

[6] "Dongwon's Tech Blog: TeraSort for Spark and Flink with Range Partitioner," [accessed 6-January-2016]. [Online]. Available: http://eastcirclek.blogspot.se/2015/06/terasort-for-spark-and-flink-with-range.html

[7] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, dec 2014, pp. 69–78. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=7027482

[8] "Benchmarking Streaming Computation Engines at... — Yahoo Engineering," [accessed 6-January-2016]. [Online]. Available: http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[9] "Karamel Lab - Repository for reproducible experiments," [accessed 6-January-2016]. [Online]. Available: https://github.com/karamel-lab

[10] "Design Experiment with Karamel — Karamel," [accessed 6-January-2016]. [Online]. Available: http://www.karamel.io/?q=content/design-experiment-karamel

[11] "Apache Storm," [accessed 6-January-2016]. [Online]. Available: http://storm.apache.org/

[12] "Highly Reliable Distributed Coordination," [accessed 6-January-2016]. [Online]. Available: https://zookeeper.apache.org

[13] "A high-throughput distributed messaging system. ," [accessed 6-January-2016]. [Online]. Available: http://kafka.apache.org

[14] "HiBench - Benchmark for Bigdata applications," [accessed 6-January-2016]. [Online]. Available: https://github.com/intel-hadoop/HiBench

[15] "Collectl - System Monitoring Tool," [accessed 6-January-2016]. [Online]. Available: http://collectl.sourceforge.net/index.html

[16] "Collectl Monitor- Tool for system level performance monitoring and reporting," [accessed 6-January-2016]. [Online]. Available: https://github.com/shelan/collectl-monitoring

[17] "Amazon EC2 Spot Instances," [accessed 6-January-2016]. [Online]. Available: https://aws.amazon.com/ec2/spot/