Apache Flink: Distributed Stream Data Processing

K.M.J. Jacobs
CERN, Geneva, Switzerland

1. Introduction

The amount of data has been growing significantly. Therefore, the need for distributed data processing frameworks is growing. Currently, there are two well-known data processing frameworks with an API for data batches and an API for data streams which are named Apache Flink [1] and Apache Spark [3]. Both Apache Spark and Apache Flink are improving upon the MapReduce implementation of the Apache Hadoop [2] framework. MapReduce is the first programming model for distributed processing on large scale that is available in Apache Hadoop.

1.1 Goals

The goal of this paper is to shed some light on the capabilities of Apache Flink by the means of two use cases. Both Apache Flink and Apache Spark have one API for batch jobs and one API for jobs based on data streams. These APIs are considered as the use cases.

In this paper the use cases are discussed first. Then, the experiments and results of the experiments are described and then the conclusions and ideas for future work are given.

2. Related work

During the creation of this report, another work on comparing both data processing frameworks has been published [6]. The work mainly focusses on batch and iterative workloads while this work focusses on batch and stream performance of both frameworks.

On the website of Apache Beam [4], a capability matrix of both Apache Spark and Apache Flink is given. According to the matrix, Apache Spark has limited support for windowing. Both Apache Spark and Apache Flink are currently missing support for (meta)data driven triggers. Both frameworks are missing support for side inputs, but Apache Flink plans to implement side inputs in a future release.

3. Applications

At CERN, some Apache Spark code has been implemented for processing data streams. One of the applications pattern recognition. In this application, one of the main tasks is de-duplication in which duplicate elements in a stream are filtered. For that, a state of the stream needs to be maintained in which unique elements are stored. The other application is stream enrichment. In stream enrichment, one stream is enriched by information of other streams. The state of the other streams need to be maintained such that it can be joined with the information of the main stream.

So in both applications, maintaining elements of a stream is important. This is investigated in the experiments. It could be achieved by iterating over all elements in the state. However, this would introduce network traffic since the elements of a stream are sent back into the stream over the network. One other solution is to maintain a global state in which all nodes can read and write to the global state. Both Apache Spark and Apache Flink have support for this solution.

4. Experiments

In this report, both the Batch and Stream API of Apache Flink and Apache Spark are tested. The performance of the Batch API is tested by means of the execution time of a batch job and the performance of the Stream API is tested by means of the latency introduced in a job based on data streams.

4.1 Reproducability

The code used for both experiments can be found on `https://github.com/kevin91nl/terasort-latency-spark-flink`. The data and the scripts for visualizing the results can be found on `https://github.com/kevin91nl/terasort-latency-spark-flink-plot`. The converted code written for Apache Flink for the stream enrichment application can be found on `https://github.com/kevin91nl/flink-fts-enrichment` and the converted code for the pattern recognition can be found at `https://github.com/kevin91nl/flink-ftsds`. The last two repositories are private repositories, but access can be requested.

4.2 Resources

For the experiments, a cluster is used. The cluster consists of 11 nodes with Scientific Linux version 6.8 installed. Every node consists of 4 CPUs and the cluster has 77GB of memory resources and 2.5TB of HDFS storage. All nodes are in fact Virtual Machines with network attached storage and all Apache Spark and Apache Flink code described in this reported is executed on top of Apache Hadoop YARN [7]. For the Stream API experiment only one computing node is used (consisting of 4CPUs and 1GB of memory reserved for the job). For that experiment, both the Apache Spark and Apache Flink job are executed locally and hence the jobs are not executed on top of YARN.

4.21 Considerations

The cluster is not only used for the experiments described in this report. Therefore, the experiments are repeated several times to filter out some of the noise. Furthermore, if too much resources were reserved, then memory errors could occur. Both Apache Flink and Apache Spark are using only 10 executors on the cluster consisting of 1GB each in order to avoid memory errors.

For testing the Stream API, only one computing node is used. This is due to the fact that since a stream is processed using a distributed processing framework, a partition of the stream is send to each node and this partition is then processed on a single node. Therefore, for measuring the latency, only a single node is sufficient.

4.3 Batch API

In order to test the Batch API of both frameworks, a benchmark tool named TeraSort [5] is used. TeraSort consists of three parts: TeraGen, TeraSort and TeraValidate. TeraGen generates random records which are sorted by the TeraSort algorithm. The TeraSort algorithm is implemented for both Apache Flink and Apache Spark and can be found in the GitHub repository mentioned earlier. The TeraValidate tool is used to validate the output of the TeraSort algorithm. A file stored on HDFS of size 100GB is sorted by the TeraSort job in both Apache Spark and Apache Flink and is validated afterwards.

4.4 Stream API

For this job, random bitstrings of a fixed size are generated and are appended with their creation time (measured in the number of milliseconds since Epoch). The job then measures the difference between the output time and the creation time. This approximates the latency of the data processing framework. For Apache Spark, a batch parameter is required which specifies the size of one batch measured in time units. This parameter is choosen such that for the configuration used in this report the approximated latency is minimal. The bitstrings are generated with the following command:

```
    < /dev/urandom tr -dc 01 | fold -w[size] | while read line; do echo
"$line" `date +%s%3N` ; done
```

Where `[size]` is replaced by the size of the bitstring.

5. Results

5.1 Batch API

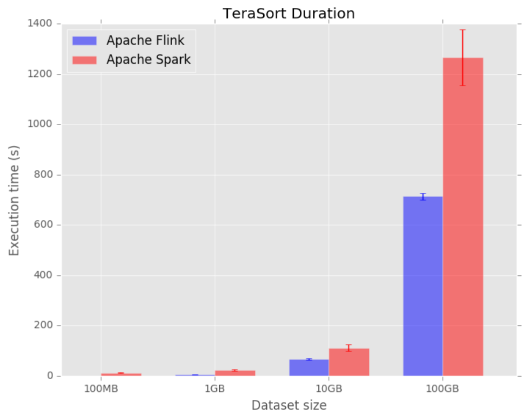
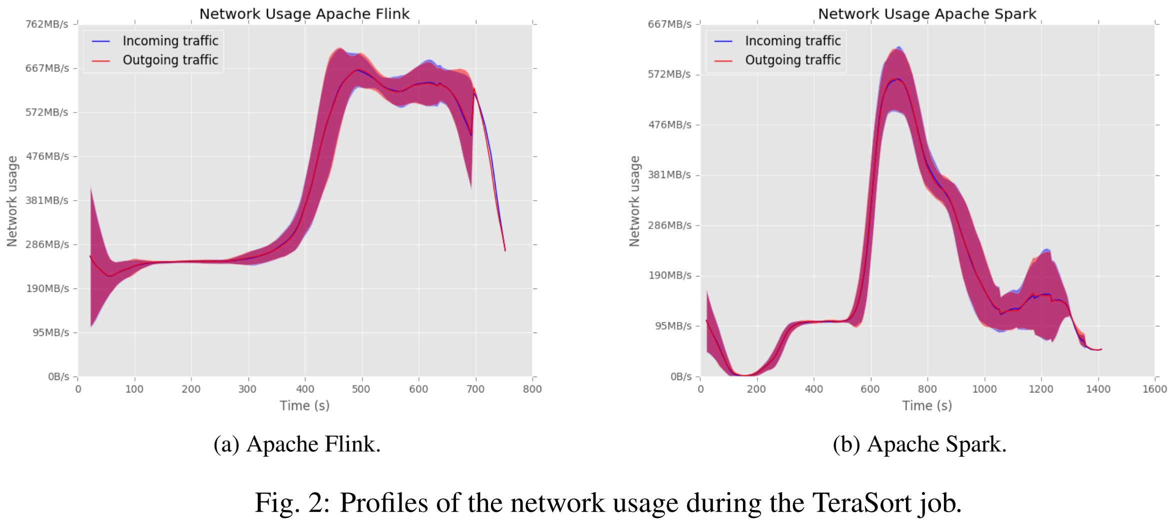


Fig. 1: The execution time of TeraSort.

In figure 1, it can be seen that Apache Flink does the TeraSort job in about half of the time of Apache Spark. For very small cases, Apache Flink almost has no execution time while Apache Spark needs a significant amount of execution time to complete the job. What also can be seen, is that the execution time of Apache Spark has a larger variability than the execution time of Apache Flink.

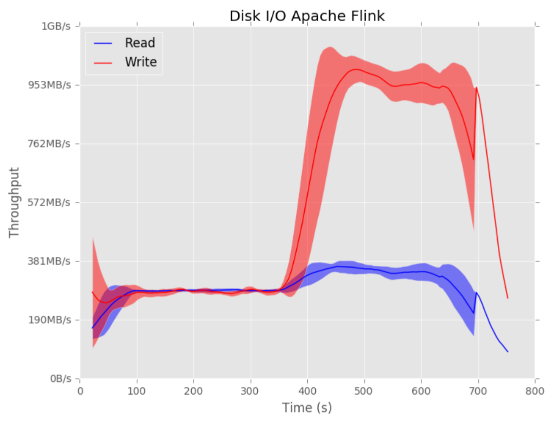
5.11 Network usage



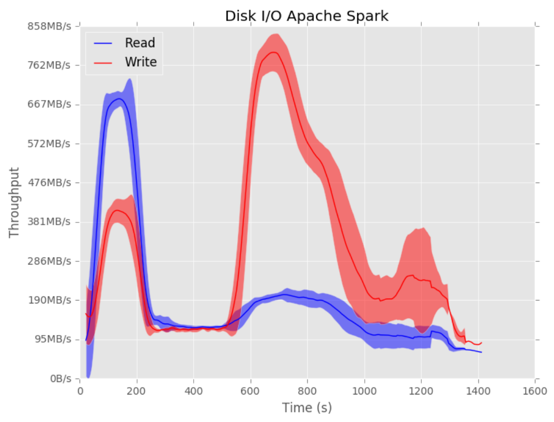
(a) Apache Flink. (b) Apache Spark.

Fig. 2: Profiles of the network usage during the TeraSort job.

From figure 2 can be seen that Apache Flink has about a constant rate of incoming and outgoing network traffic and Apache Spark does not have this constant rate of incoming and outgoing network traffic. What also can be seen from the graphs, is that the data being sent over the network and the same amount of data is received. This is due to the fact that the monitoring systems monitor the entire cluster.



(a) Apache Flink.



(b) Apache Spark.

Fig. 3: Profiles of the disk usage during the TeraSort job.

5.12 Disk usage

It is important to notice that the graph of the disk usage (figure 3) is very similar to the graph of the network usage (figure 2). This is due to the fact that the disks are in fact network attached storage. By that, the behaviour of the disk also reflects to behaviour of the network. Apache Spark is almost not using the network at the beginning of the job which can be seen from figure 2. In figure 3 can be seen that Apache Spark is then reading the data from disk.

5.2 Stream API

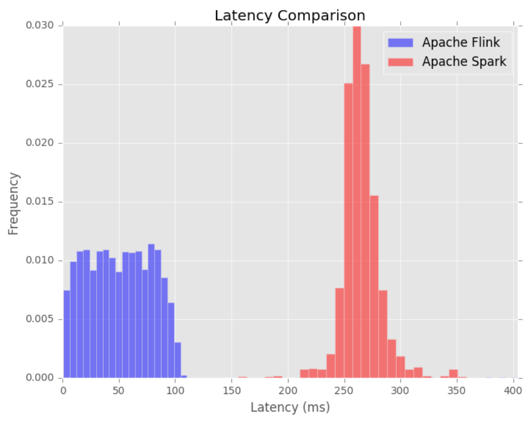


Fig. 4: The latency introduced by Apache Spark and Apache Flink.

The fact that Apache Flink is fundamentally based on data streams is clearly reflected in 4. The mean latency of Apache Flink is $54ms$ (with a standard deviation of $50ms$) while the latency of Apache Spark is centered around 274ms (with a standard deviation of 65ms).

6. Conclusions

The obtained results for the batch experiment are valid only in computing clusters with network attached storage.

Apache Flink is fundamentally based on data streams and this fact is reflected by the latency of Apache Flink (which is shown in figure 4). The latency of Apache Flink is lower than the latency of Apache Spark.

Apache Flink is also better in terms of batch processing using the configuration described in this report. The behaviour of resource usage of Apache Flink and Apache Spark differs by the fact that Apache Flink is reading from disk constantly and Apache Spark is reading most of the data in the beginning. This results in a predictable execution time and it can be seen that the variability of Apache Spark jobs is larger than the variability of Apache Flink jobs (which is reflected in figure 1).

Besides these facts, the API of Apache Spark is limited compared to the API of Apache Flink.

Combining the results, Apache Flink is a good candidate for replacing Apache Spark.

7. Future work

It would be interesting to see the difference in Apache Spark and Apache Flink on other configurations. Besides that, an Apache project which is called Apache Beam is meant to fix the shortcommings of both data processing frameworks. It is also worth investigating in Apache Beam when it is released since it is intended to have advantages of both data processing frameworks and implement functionality that currently is missing in both data processing frameworks.

References

[1] Apache Flink. `http://flink.apache.org/`. Accessed: 2016-06-27.

[2] Apache Hadoop. `http://hadoop.apache.org/`. Accessed: 2016-07-06.

[3] Apache Spark. `http://spark.apache.org/`. Accessed: 2016-06-27.

[4] Capability Matrix. `http://beam.incubator.apache.org/learn/runners/capability-matrix/`. Accessed: 2016-08-15.

[5] TeraSort for Apache Spark and Apache Flink. `http://eastcirclek.blogspot.ch/2015/06/terasort-for-spark-and-flink-with-range.html`. Accessed: 2016-07-06.

[6] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S. Pérez. Spark versus flink: Understanding performance in big data analytics frameworks. *IEEE 2016 International Conference on Cluster Computing*, July 2016.

[7] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.