SPARK VERSUS FLINK: PEFORMANCE COMPARISONS IN BIG DATA ANALYSIS FRAMEWORK

April 17, 2019

Xueying Li
Shayan Manoochehri

Abstract Spark and Flink are two Apache-hosted data analytic frameworks that facilitate the analysis of streaming datasets. In-depth understanding of the underlying architecture of any framework is important to increase the performance of processing data. As such this work is aimed to study the building blocks of Spark and Flink and to evaluate their performance by processing streaming datasets; different benchmark applications will be considered for this evaluation. Fault tolerance, a major aspect of stream processing will be discussed, and support for applications such as Online Machine Learning and real time Graph Analysis using stream data processing will be examined.

Introduction

Stream computing for real-time analytic is an important field in the topic of big data. When it comes to streaming data, we can not avoid the importance of the two most powerful data processing engines: Flink and Spark.Streaming data is the data that comes in continuously. The typical use case of streaming data could be querying tweets in real time to get insights of trends in social network; we use this use case in our benchmark applications. Other streaming data examples are traffic, stock and weather.

To outperform the original platforms' shortcomings, both Flink and Spark try not to persist their data to storage but keep in-memory as far as possible. From the design perspective, however, Flink was originally built for streaming whereas Spark was to replace the batch-oriented Hadoop system; Apache Spark Streaming is part of the ecosystem. Even though Flink and Spark look similar in syntax of the programs, they have main differences. Flink is build from the ground up for streaming processing while Spark added streaming to its ecosystem, which similar to Hadoop, runs over static datasets. In Flink, if the input data stream is bounded, its behavior resembles batch processing[1].

With respect to streaming data, Spark continuously iterates through streaming data to divide them into micro batches. Flink uses checkpoints to mark finite datasets wherein streams are not necessarily opened and closed. As we will discuss later, processing live data always cause latency, i.e., the reducer operation will run on map datasets which was previously created.To run static datasets, Flink processes data the same regardless of being finite or not. But Spark uses Discretized Streams (DStream) for streaming data and Resilient Distributed Dataset (RDD) for batch data[2].

In the following, we first provide direct and in-depth comparison between Spark and Flink with regard to processing streaming through two benchmark applications as follows comparing their respective latency and throughput. There are other differences between Spark and Flink including fault tolerance and dealing with loops in streaming applications such as Online Machine Learning and real time Graph Analysis which will be compared in fault tolerance and application section*s[3].

Benchmark

Our streaming data source is the stream of tweets is extracted using a twitter API[4]. In our applications we utilize a subset of tweet's fields: id, date tweet created, user, language , re-tweet count and favourite count.

We developed two applications for each Flink and Spark Steaming. In the first one we are searching for the maximum/minimum number of tweets for each language within a time interval, e.g., we could be interested to know which language people have written tweets the most/least in each 2, 6, 12 second intervals. In doing so we first require to calculate the number of tweets for each language in each time window and then extract the one with maximum/minimum number of tweets.

The second application is searching for the ten most popular hash tags used in each time interval. This involves parsing each tweet to extract the hash tags and then similar to the first application we first compute the number of tweets for each hash tag and finally identify the most popular hash tags. In the following we discuss the specifics for each of the frameworks in more details.

Flink

Flink is a true streaming system and as such time should be precisely defined in. As shown in the Figure 1 there are three different type of time: event, ingestion and processing time. Event time is crucial to have the deterministic results regardless of the order of receiving the events. We set the environment setting to work with event times for our applications wherein creation date/time of each tweet is set as event time.

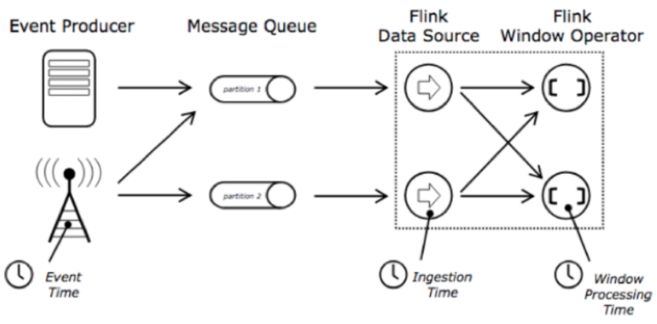


Figure 1: Three different types of time

Along with event times we also need to decide on generation of watermarks. Watermarks are required to control how long a window operator needs to wait for new data to arrive before starting the computation, e.g., to trigger a window computation, emit the result or evict the content of a window. In this sense watermarks balance the latency and accuracy of the result for the date in hand. However in practice we never can guarantee the accuracy due to network disconnections so in Flink the correction of result is performed by allowing the updates as late events arrive.

The kind of watermarks we use is periodic as opposed to punctuated which is data dependent. Once the watermark passes the end of a window the computation is triggered which is the default behavior.After defining the event time and watermark, the calculation performed on windows are specified. This is the duty of the window functions in order to process the elements of each window as soon as the system determines the appropriate time to do so. Generally there are two types of such functions which differ in performance and flexibility.

The first type which can be seen in functions such as reduce, aggregate and fold are quite fast due to incrementally aggregating the elements for each window as they arrive. The other type seen in functions such as process window functions have more flexibility for defining the business logic but lack the performance as they have to acquire all the elements of a window before performing the computation. Moreover a context object with access to time and state information is handed to these functions which we used in our implementation to compute latency.

We calculate the latency by subtracting window end time from the of the processing time for the same window. Sometimes however we can combine both of these types of function to benefit from the performance of the former and flexibility of the latter.

Having the latency computed for each window intervals we plot our data with x-axis as progress time and the y-axis for both latency and throughput computed for each window. This is done for the spark streaming as well to conclude our findings regarding their respective performance.

Spark Streaming

Spark Streaming is built on the Discretized Stream or DStream construct which represent data stream flowing from source throughout transforming operators. DStream consists of a unbounded series of RDDs, Spark's abstraction for immutable, distributed datasets. The idea in DStreams is to structure a streaming computation as a series of stateless, deterministic batch computations on small time intervals, i.e., performing transformation operations on DStreams translates to applying the same operations on underlying RDDs.

For a Spark Streaming application, the system should be capable to process data to keep up with data generation. In other words, batches of data should be processed as fast as they are being generated. This requires to have the batch processing time to be less than the batch interval. Setting the right size of batch interval required us some monitoring effort to find the best performance using our tweet data stream, a disadvantage we experienced in Spark Streaming versus Flink.

Similar to Flink, Spark Streaming also provides windowed computations, such as different kinds of reduce by window operations such as keyed or non-keyed ones. This allow us to apply transformations over a window of arriving data. Any window operation needs to specify two parameters. i) window length: the duration of the window, ii) sliding interval: the interval at which the window operation is performed. Both of these parameters are set to same value in our code as we don't require to support sliding windows. These two parameters must be multiples of the batch interval of the source DStream.

Experimental Results

For the experiments we used a Macbook Pro machine with 2.7GHz dual-core Intel Core i5 processor, 3MB shared L3 cache and 8 GB 1867 MHz DDR3. We used Spark v2.4 and Flink v1.8 in our implementation code; the source code can be found at https://github.com/ShaneMann/flink-spark-streaming. After fine tuning with Spark applications we chose batch intervals of 2 seconds for all of its executions.The results are depicted in the diagrams (Figure 2 to 5) showing the latency and throughput for each application for different time windows: 2,6, and 12 seconds.

As shown in diagrams (Figure 2 and 3) the latency is quite lower and more stable in Flink applications compared to Spark Streaming results. The lower latency leads to have the higher in both Flink applications. Spark applications however show large fluctuations for the same intervals with higher latency.This result makes sense as Flink acts on an event as soon as it is received while Spark needs to make a micro batches of events before starting the computation.

Figure 4a/b and 5a/b shows the results of throughout for the same time intervals mentioned above. For 2 second time window spark performs similar to Flink, however, as the time windows increases Flink outperforms Spark Streaming. This is expected as Spark will not start the window computation until it received all the elements leading to longer processing time and lower throughput.

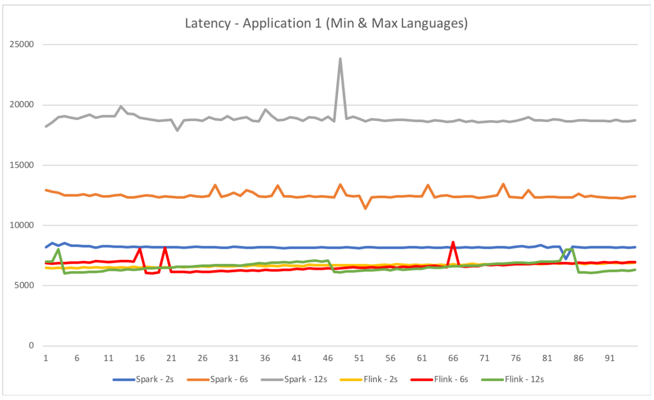


Figure 2: Latency comparison

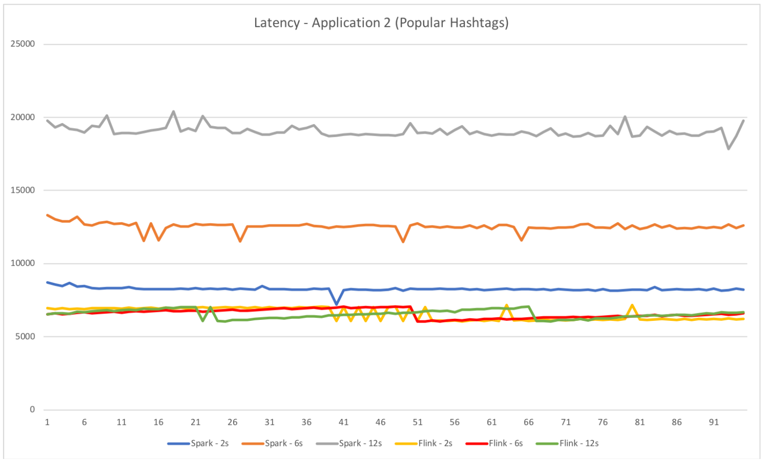
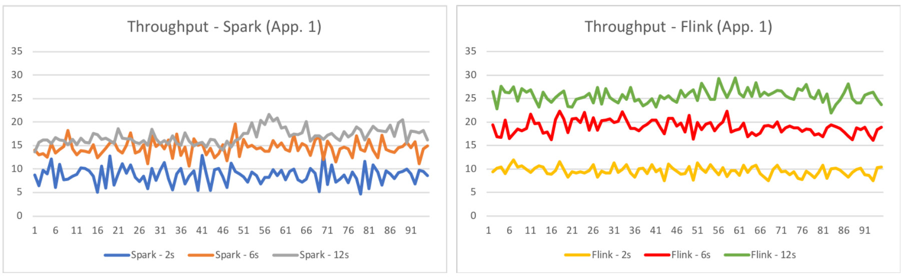
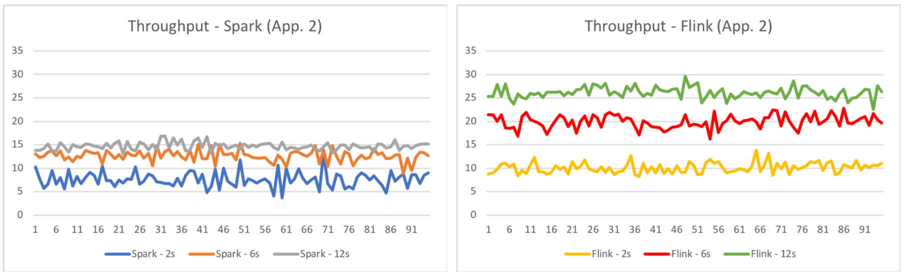


Figure 3: Latency comparison



(a) Spark **(b)** Flink

Figure 4: Throughput comparisons



(a) Spark **(b)** Flink

Figure 5: Throughput comparisons

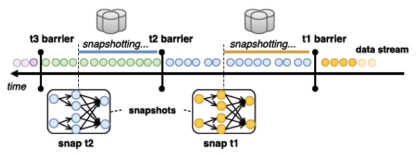
Fault Tolerance

Flink

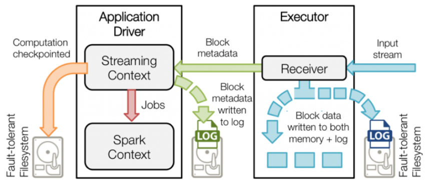
Fault Tolerance in Flink is realized through checkpointing recovery mechanism via distributed consistent snapshots along with partial re-execution. Using such mechanism the exactly-once-processing is guaranteed due to the durable, re-playable data sources.

At regular intervals of checkpointing consistent snapshots of all operators states, including the current position of the input streams are acquired. Such snapshots will be used during recovery in face of failures. Additionally, partial recovery of a failure is possible by partial re-execution of unprocessed records buffered at the immediate upstream operators.

Naive approach for checkpointing is to periodically stall the overall computation so as to take global state snapshots. This however increase the latency and leads to storage of larger than required snapshots. Flink uses a mechanism called Asynchronous Barrier Snapshotting (ABS)[5], a lightweight algorithm to minimise space requirements of stateful stream processing. This is realized by barriers, used as control records, injected into the input streams corresponding to a logical time in order to rationally partition the stream data flow. Therefore, the snapshot of all operator states on acyclic execution topologies correspond to the same logical time in the computation.



(a) ABS of Flink



(b) Writing ahead logs of Spark

Figure 6: Fault Tolerance

In this mechanism, when an operator receives barriers it first makes sure that the barriers from all inputs have been received; this is referred to as alignment phase. Once the state has been written to a durable storage, the operator forwards the barrier to next operator(s) downstream. Only when, all operators registered a snapshot of their state a global snapshot is complete. E.g., Figure 6 shows snapshot t1 and t2 contain all operator states that are the result of processing all records before t1 and t2 barriers respectively.In case of failure recovery process will restores the states of all operators to last snapshot and resets the input streams to the corresponding barrier used in this snapshot.

Spark Streaming

Initially, spark streaming used writing ahead logs as fault tolerance mechanisms shown as Figure 4b. However, a drawback is that replication could not be avoided. Later this design improved by RDDs keeping data in memory with best effort and recovering from failures without replication by tracking the lineage graph of operations.

Spark Streaming, built on DStreams, provides fully deterministic and visibility at a fine granularity of state at each time-step operation. This provides us strong foundation of a powerful recovery mechanisms which outperform replication and upstream backup: parallel recovery of a lost node's state. In this recovery process, when a node fails, each node in the cluster works collaboratively with other nodes to re-compute part of the lost node's RDDs. This results in considerably faster recovery than upstream backup without the cost of replication.

Applications

Both real time Graph Analysis and Machine Learning applications can benefit from stream processing frameworks: incremental processing. Besides, processing of endless live data is the main characteristics of such applications. For example, we may train a machine learning model as data arrives in response to changing patterns to keep up with near real time predictions in areas such as domestic news, social networks or trends in financial markets.

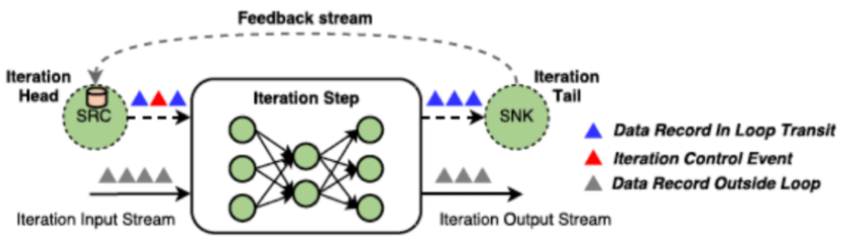


Figure 7: Iterations in Flink

General approaches to support iterations involves either submitting a new job for an iteration, appending additional nodes to the DAG at runtime, or including feedback edges.Iterations in Flink are implemented by special operators containing an execution graph of iteration steps, head and tail tasks which are implicitly connected with feedback edges; this provides asynchronous stream iterations with no need to coordination (Figure 7). Spark, on the other hand, is falling behind to support these types of applications due to lower performance[6][7].

Conclusion

In this work, we compared streaming computation of two most popular frameworks Spark and Flink with respect to computation performance and some major aspects of structural designs. A Benchmark was designed wherein two applications implemented to compare the latency and throughput for performance comparison. Our result showed Flink has stable, lower latency and higher throughput as compared to Spark Streaming. We also presented the fault tolerance mechanisms of both frameworks. Finally we discussed the loop implementation in Flink followed by its support for major streaming applications: real time Graph Analysis and Online Machine Learning.

References

[1] Carbone, Paris, et al. *Apache flink: Stream and batch processing in a single engine.*Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36.4, 2015.

[2] Alex Kendall, Vijay Badrinarayanan, Roberto Cipolla *In Search of Data Dominance: Spark Versus Flink* PAMI, 2017.

[3] Mail Dataset *https://training.ververica.com/exercises/mailData.html*

[4] Twitter For Developer *https://developer.twitter.com/*

[5] P. Carbone, G. F´ora, S. Ewen, S. Haridi, and K. Tzoumas. *Lightweight asynchronous snapshots for distributed dataflows.* arXiv:1506.08603, 2015.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. *HaLoop: Efficient Iterative Data Processing on Large Clusters.* PVLDB, 2010.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster Computing with Working Sets.* USENIX HotCloud, 2010.

[8] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. *Naiad: a timely dataflow system.* ACM SOSP, 2013.

[9] Marcu, Ovidiu-Cristian, et al. *Spark versus flink: Understanding performance in big data analytics frameworks.* IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2016.

[10] Fabian Hueske, Vasiliki Kalavri *Stream Processing with Apache Flink Fundamentals, Implementation, and Operation of Streaming Applications* April 2019.

[11] Fabian Hueske, Vasiliki Kalavri *Stream Processing with Apache Spark* June 2019.

[12] Chintapalli, Sanket, et al. *Benchmarking Streaming Computation Engines at Yahoo!.* Tech. Rep. (2015).

[13] Apache Flink vs. Apache Spark *https://dzone.com/articles/apache-flink-vs-apache-spark-brewing-codes/*

[14] Apache Showdown: Flink vs. Spark *https://jobs.zalando.com/tech/blog/apache-showdown-flink-vs.-spark/*