SOEN6761
SPARK VERSUS FLINK: PEFORMANCE COMPARISONS IN BIG DATA ANALYSIS FRAMEWORK

April 17, 2019

Shayan Manoochehri 27232438
Xueying Li 40036265
SOEN 691 Big Data Project Report

Abstract Spark and Flink are two Apache -hosted data analytic frameworks that facilitate the analyzing of big datasets. The in-depth understanding of the underlying architecture choices are important to increase the performance of processing data with respect to different datasets. This project is aimed to justify the performance of Spark and Flink by evaluating their results by processing on streaming datasets; different benchmarks will be considered for this evaluation. Fault tolerance, a major aspect of stream processing will be discussed, and support for applications such as Machine Learning with respect to stream processing will be discussed.

0.1 Introduction

Streaming computing and real-time analytic are two important fields in the topic of big data. When it comes to streaming data, we can not avoid the influential 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 is querying tweets in real time. Other streaming data examples are traffic, stock and weather. The way Flink and spark works is that they do not persist their data to storage but keep in-memory and use it right away[1].

From the design perspective, Apache Flink was originally built for streaming whereas Apache Spark was to replace the batch-oriented Hadoop system; Apache Spark Streaming is part of the ecosystem. Even though Flink and Spark look similar, they have main differences. Flink is build from the ground up for streaming processing while Spark added streaming to their ecosystem, which similar to Hadoop, runs over static datasets. In Flink, if the input data stream is bounded, its behavior is similar to batch processing[2].

With respect to streaming data, Spark continuously iterates through streaming data to divide them into micro batches. Whereas Flink uses checkpoints to mark finite datasets; streams are not necessarily opened and closed. And as we will discuss later, processing live data always cause latency. In other words, the reducer operation will run on map datasets which was just created before.

To run static datasets[8], Flink processes data the same regardless of being finiteor not. But for Spark, Discretized Streams (Dstream) are used for streaming data and Resilient Distributed Dataset (RDD) for batch data. Note that there are differences between Spark and Flink including framework design, APIs, fault tolerence mechanism and Machine Learning which we will compare in this work. Moreover, we will provide direct and in-depth comparison between Spark and Flink with regard to processing streaming datasets.

0.2 Benchmark Design

Our streaming data sources are the tweets which are extracted using Twitter API's[7]. For the applications that we developed we work with the 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 application we are searching for the maximum number of tweets for each language for each time interval,e.g., we could be interested to know which language people have written tweets in each 30 seconds intervals. In doing so we first require to calculate the number of tweets for each language for each time window and then extract the one with maximum number of tweets.

The second application is searching for the 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 need to compute the number of tweets for each hash tag and finally recognize the most popular hash tags.

In the following we discuss the specifics for each of the frameworks in more details.

Flink Streaming

Flink is the a true streaming system and as such time should be precisely defined in it. As shown in the figure above 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. As such we set the environment setting to work with event times for our applications. Moreover we set creation date and time of each tweet as event time.

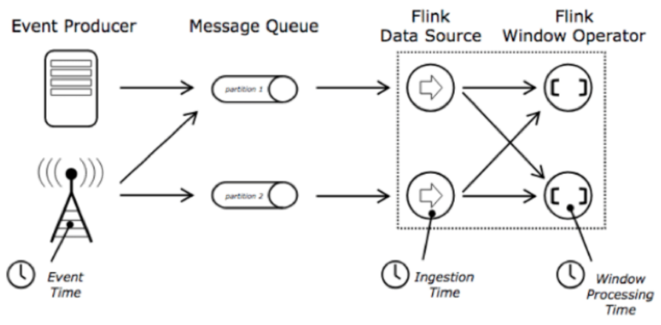


Figure 1: Three different types of time

Along with event times we decide on generation of watermarks.Watermarks are required to control how long to wait for data to arrive before computation starts,e.g., to trigger a time window computation, emit the result or evict the content of a window. As such they can 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 decided upon 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, we need to indicate the calculation performed on time windows. This is the duty of the window functions, which are used 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 kind such as reduce, aggregate and fold functions are fast as they can incrementally aggregate the elements for each window as they arrive. The other 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: the processing time minus window end time.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 as latency. This is done for the spark streaming as well to compare 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 to transforming operators.DStream consists of a unbounded series of RDDs as Sparks abstraction of immutable, distributed dataset.The idea in D-Streams 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[1].

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 however requires some monitoring of different intervals using our tweet data stream, a disadvantage we experienced in Spark vs. Flink Streaming.

Similar to Flink,Spark Streaming also provides windowed computations, such as reduce or by window operations. This allow us to apply transformations over a time window of 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 have sliding windows. These two parameters must be multiples of the batch interval of the source DStream.

0.3 Experimental Results

Following figures show the experimental results of comparison in terms of the designed two application. It is found from Figure 2 that the latency time for Flink is more stable when window is set from 2s to 12s while Spark changed in a wide range from 10000ms to 20000ms with the same time interval. It is shown that Flink performs better than Spark as its latency was always less than that of Spark.

Following given the results of Throughout in terms of different window time(time interval)

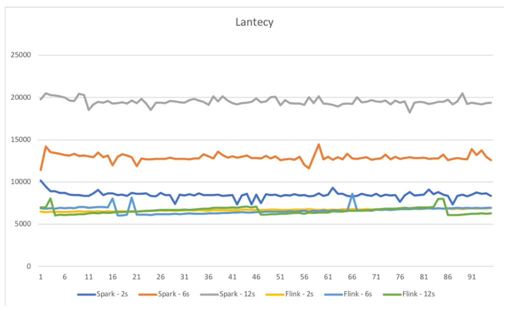
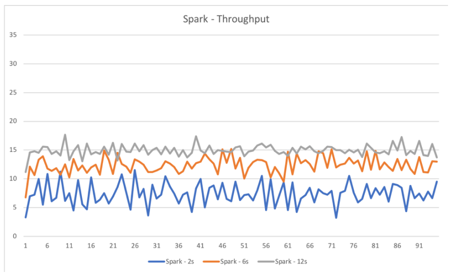
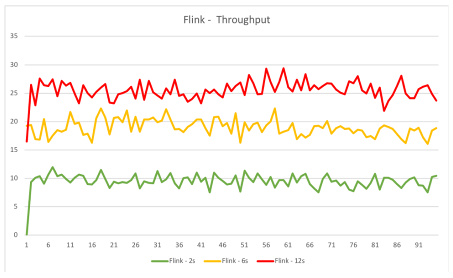


Figure 2: Latency comparison

from 2s to 12s. As we could see, spark performs better than Flink. And the increase rate with respect of the increase of time interval for Flink is higher than spark.



(a) Throughput of Spark



(b) Throughput of flink

Figure 3: Throughput comparisons

From the experimental results, Flink shows a better performance of latency while throughput of Spark is less.

0.4 Fault Tolerance

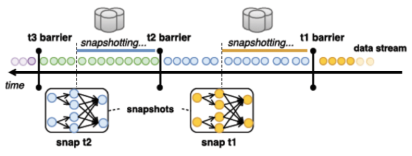
Flink Streaming
Fault Tolerance in Flink is realized through checkpointing and recovery via distributed consistent snapshots and partial re-execution wherein the exactly-once-processing is guaranteed; this is possible due to the durable, re-playable data sources.

At regular intervals 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.

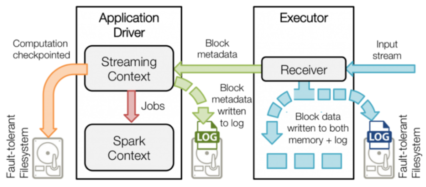
Naive approach for checkpointing is to periodically stall the overall computation so as to take global state snapshots.This however increase the latency and leads to storage of larger than required snapshots. Flink uses a mechanism called Asynchron ous Barrier Snapshotting (ABS[3]), a lightweight algorithm to minimise space requirements of stateful stream processing. This is realized by barriers, used as control records, injected into the input streams that correspond to a logical time and logically 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. On cyclic data flows Flink only keeps a minimal record log.

When an operator receives barriers 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 below shows snapshot t1 and t2 contains 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.



(a) ABS of Flink



(b) Writing ahead logs of Spark

Figure 4: Fault Tolerance

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. Then, RDDs is improved to keeps data in memory and can recover it without replication by tracking the lineage graph of operations that were used to build it. Spark Streaming,built on D-Streams, provides fully deterministic and visible at a fine granularity of state at each time-step operation. This enables powerful recovery mechanisms which outperform replication and upstream backup: parallel recovery of a lost node's state. In this mechanism, when a node fails, each node in the cluster works to re-compute part of the lost node's RDDs. This results in considerably faster recovery than upstream backup without the cost of replication.

0.5 Applications

Both real time graph analysis and machine learning applications can benefit from stream processing frameworks: Incremental processing, i.e., Iterations and processing of endless live data are the main characteristics of such applications. For example, we may train a machine learning mode as data arrives in response to changing patterns to keep up with close to real time predictions in areas such as domestic news, social networks or trends in financial markets.

General approaches to support iterations involves either submitting a new job for an iteration, appending additional nodes to the DAG[4][5] at runtime, or including feedback[6] edges.

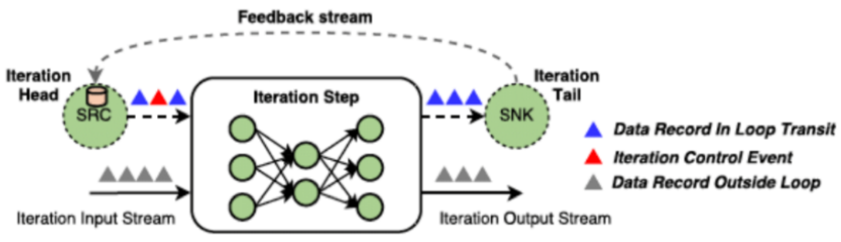


Figure 5: Iterations in Flink

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 a synchronous stream iterations with no need to coordination(Figure5).Spark on the other hand is falling behind to efficiently support say online learning applications.

0.6 Conclusion

In this paper, we present the comparisons of streaming computation of two most popular frameworks, Spark and Flink, in terms of computation performance and structured designs. An experiment with two applications was implemented to compare the latency and throughput of the two frameworks. And it is concluded that Flink shows less latency while Spark shows less throughput. Also, we compared the fault tolerance mechanisms and it is shown that Flink shows better design as its performance is decoupled from other sending messages while that of Spark highly depended on writing to write ahead logs. And we also analyzed the iteration steps of Flink in machine learning algorithms. For future improvement, more experiments should be designed and implemented to compare the static performance of the two frameworks.

References

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

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

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

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

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

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