2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming

Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh
Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng and Paul Poulosky
Yahoo Inc.
{schintap, derekd, evans, rfarivar, tgraves, holder, zhuol, knusbaum, kpatil, jerrypeng, ppoulosk}@yahoo-inc.com

***Abstract*—Streaming data processing has been gaining attention due to its application into a wide range of scenarios. To serve the booming demands of streaming data processing, many computation engines have been developed. However, there is still a lack of real-world benchmarks that would be helpful when choosing the most appropriate platform for serving real-time streaming needs. In order to address this problem, we developed a streaming benchmark for three representative computation engines: Flink, Storm and Spark Streaming. Instead of testing speed-of-light event processing, we construct a full data pipeline using Kafka and Redis in order to more closely mimic the real-world production scenarios. Based on our experiments, we provide a performance comparison of the three data engines in terms of 99th percentile latency and throughput for various configurations.**

***Keywords*-Streaming processing, Benchmark, Storm, Spark, Flink, Low Latency**

I. Introduction

Streaming data processing has been attracting more and more attention due to its crucial impacts on a wide range of use cases, such as real-time trading analytics, malfunction detection, campaign, social network, smart advertisement placement, log processing and metrics analytics. To serve the booming demands of streaming data processing, many computation engines have sprung up. There are now several engines that are widely adopted, including Apache Storm [1], Apache Flink [2], Apache Spark (Spark Streaming) [3], [4], Apache Samza [5], Apache Apex [6] and Google Cloud Dataflow [7] among others. For example, at Yahoo, the platform of choice has been Apache Storm, which replaced the internally developed S4 platform [8]. JStorm [9] (now being merged into Apache Storm) and Heron [10] are heavily used by Alibaba Inc. and Twitter Inc., respectively, while Spark Streaming and Flink are also gaining widespread adoption.

However, there is increasing confusion over which package offers the best set of features and which one performs better under which conditions. Researchers have invested effort in comparing the features and performance of these popular streaming processing platforms [11], [12], [13]. Most of this work focuses on feature comparisons and throughput/latency evaluation through speed-of-light tests. However, there is still a lack of a real-world streaming benchmark that would help users to make realistic and comprehensive comparisons among different computation engines.

In order to help users to choose a most appropriate platform for serving their big data real-time streaming needs, we designed and implemented a real-world streaming benchmark and released it as open source [14]. In this benchmark, we bring in Kafka [15] and Redis [16] for data fetching and storage to construct a full data pipeline to more closely mimic the real-world production scenarios. In our initial evaluation we focus on three of the most popular platforms (Storm, Flink and Spark), which show different advantages and shortcomings. The results demonstrate that at fairly high throughput, Storm and Flink have much lower latency than Spark Streaming (whose latency is proportional to throughput rate). On the other hand, Spark Streaming is able to handle higher maximum throughput rate while its performance is quite sensitive to the batch duration setting.

II. Benchmark Design

Our streaming benchmark simulates an advertisement analytics pipeline. In this pipeline, there are a number of advertising campaigns, and a number of advertisements for each campaign. The job of the benchmark is to read various JSON events from Kafka, identify the relevant events, and store a windowed count of relevant events per campaign into Redis. These steps attempt to probe some common operations performed on data streams.

The flow of operations is as follows (shown in Fig. 1):

1) Read an event from Kafka.
2) Deserialize the JSON string.
3) Filter out irrelevant events (based on event_type field)
4) Take a projection of the relevant fields (ad_id and event_time)
5) Join each event by ad_id with its associated campaign_id. This information is stored in Redis.
6) Take a windowed count of events per campaign and store each window in Redis along with a timestamp of the time the window was last updated in Redis. This step must be able to handle late events.

The input data has the following schema:

- user_id, page_id, ad_id: UUID
- ad_type: String in banner, modal, sponsored-search, mail, mobile

Authors are listed in alphabetical order of last names.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.138

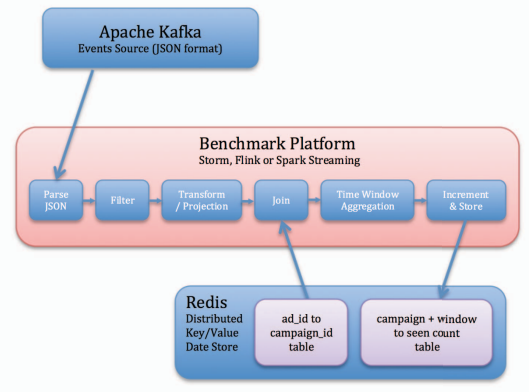


Fig. 1: Streaming Benchmark Design.

- event_type: String in view, click, purchase
- event_time: Timestamp
- ip_address: String

Producers create events with timestamps marking creation time. Truncating this timestamp to a particular digit gives the begin-time of the time window the event belongs in. In Storm and Flink, updates to Redis are written periodically, but frequently enough to meet a chosen SLA. Our SLA was 1 second, so once per second we wrote updated windows to Redis. Spark operated slightly differently due to great differences in its design. There are more details on that in the Spark section. Along with the data, we record the time at which each window in Redis was last updated as *last_updated_at*. After each run, a utility reads windows from Redis and compares the windows' start times to their *last_updated_at* times, yielding a latency data point. We calculate the event latency by subtracting the window start time and duration time from the *last_updated_at* time.

Since the Redis node in our architecture only performs in-memory lookups using a well-optimized hashing scheme, it will not become a bottleneck. The nodes are homogeneously configured, each with two Intel E5530 processors running at 2.4GHz and a total of 16 cores (8 physical, 16 hyperthreading) per node. Each node has 24GB of memory, and the machines are all located within the same rack, connected through a gigabit Ethernet switch. The cluster has a total of 40 nodes available, out of which up to 30 are used in this benchmark. For all three computation engines, we configure the platform with 10 worker nodes and one coordination node such as Storm's Nimbus or Spark's master. There are 5 Kafka nodes with 5 data partitions, 1 Redis node and 3 ZooKeeper nodes.

We ran up to 10 instances of the Kafka producers to create the required load since individual producers begin to fall behind at around 17,000 events per second. In total, we use anywhere between 25 to 30 nodes in this benchmark. The use of 10 worker nodes for a topology is near the average number we see being used by topologies internal to Yahoo. In the benchmark we run 100 campaigns, with 10 ads per campaign.

At the beginning, the benchmark's Kafka is cleared, Redis is populated with initial data (ad_id to campaign_id mapping), the streaming job is started, and then after a bit of time to let the job finish launching, the producers are started with instructions to produce events at a particular rate, giving the desired aggregate throughput. The system was left to run for 30 minutes before the producers were shut down. A few seconds were allowed for all events to be processed before the streaming job itself was stopped. The benchmark utility was then run to generate a file containing a list of $last\_updated\_at - start\_time$ numbers. These files were saved for each throughput we tested and then were used to generate the charts in this document. This benchmark is implemented in three different versions, respectively for Flink, Spark and Storm. In Section III, IV and V, we will describe each implementation and its results in details.

III. Flink

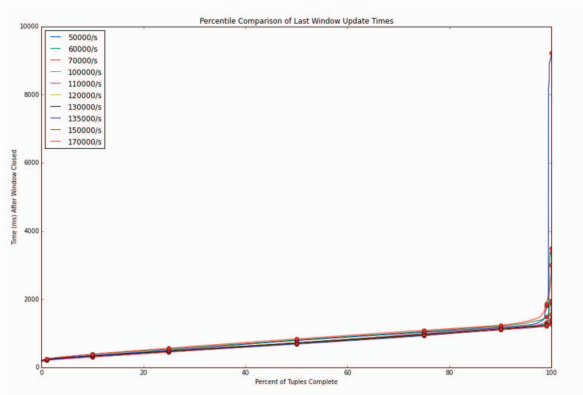


Fig. 2: Flink Performance.

The benchmark for Flink was implemented in Java by using Flink's DataStream API. The Flink DataStream API has many similarities to Storm's streaming API. For both Flink and Storm, the dataflow can be represented as a directed graph. Each vertex is a user defined operator and each directed edge represents a flow of data. Storm's API uses spouts and bolts as its operators while Flink uses map, flatMap, as well as many pre-built operators such as filter, project, and reduce. Flink uses a mechanism called checkpointing to guarantee processing which offers similar guarantees to Storm's acking. Flink has checkpointing off by default and that is how we ran this benchmark. Notable configurations we used in Flink are taskmanager.heap.mb as 15360 and taskmanager.numberOfTaskSlots as 16.

The Flink version of the benchmark uses the FlinkKafka-Consumer to read data in from Kafka. The JSON-formatted data is read from Kafka, and is then deserialized and parsed by a custom-defined flatMap operator. Once deserialized, the data is filtered via a custom-defined filter operator. Afterwards, the filtered data is projected by using the project operator. From there, the data is joined with data in Redis by a custom-defined flapMap function. Lastly, the final results are calculated from the data and written to Redis.

The rate at which Kafka emitted data events into the Flink benchmark is varied from 50,000 events/sec to 170,000 events/sec. For each Kafka emit rate, the percentile latency for

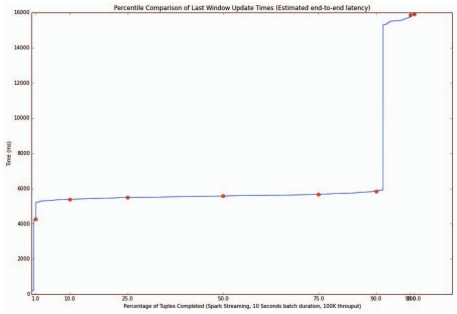
a tuple to be completely processed in the Flink benchmark is illustrated in the Fig. 2.

The percentile latency for all Kafka emit rates are relatively the same. The percentile latency rises linearly until around the 99th percentile, where the latency appears to increase exponentially.

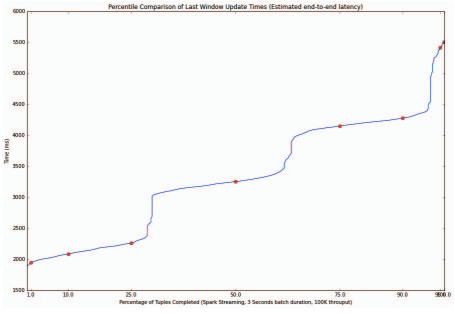
IV. Spark Streaming

For Spark benchmark, the code was written in Scala. Since the micro-batching methodology of Spark Streaming is different than the pure streaming nature of Storm, we needed to rethink parts of the benchmark. Storm and Flink benchmarks would update the Redis database once a second to try and meet our SLA, keeping the intermediate update values in a local cache. As a result, the batch duration in the Spark streaming version was set to 1 second, at least for smaller amounts of traffic. We had to increase the batch duration for higher throughput. The benchmark is written in a typical Spark style using DStreams. DStreams are the streaming equivalent of regular RDDs, and create a separate RDD for every micro batch.

It should be noted that our writes to Redis were implemented as a side-effect of the execution of the RDD transformation in order to keep the benchmark simple, so this would not be compatible with exactly-once semantics. We found that with high enough throughput, Spark Streaming was not able to keep up if it does not change batch interval setting. At 100,000 messages per second the latency greatly increased.



(a) 10 seconds batch duration



(b) 3 seconds batch duration

Fig. 3: Spark Streaming Performance.

The final results are interesting. There are essentially three behaviors for a Spark workload depending on the window duration. First, if the batch duration is set sufficiently large, the majority of the events will be handled within the current micro batch. Fig. 3(a) shows the resulting percentile processing graph for this case (100K events, 10 seconds batch duration).

But whenever 90% of events are processed in the first batch, there is possibility of improving latency. By reducing the batch duration sufficiently, we get into a region where the incoming events are processed within 3 or 4 subsequent batches. This is the second behavior, in which the batch duration puts the system on the verge of falling behind, but is still manageable, and results in better latency. This situation is shown Fig. 3(b) for a sample throughput rate (100K events, 3 seconds batch duration).

If the batch duration is reduced any more, Spark streaming falls behind. In this case, the benchmark takes a few minutes after the input data finishes to process all of the events. Under this undesirable operating region, Spark spills lots of data onto disks.

One final note is that we tried the new back pressure feature introduced in Spark 1.5. If the system is in the first operating region, enabling back pressure does nothing. In the second operating region, enabling back pressure results in longer latencies. Our experiments showed that the current back pressure implementation did not help our benchmark, and as a result we disabled it.

V. Storm

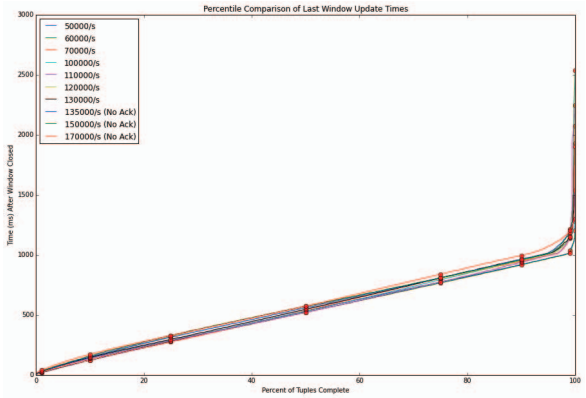


Fig. 4: Storm 0.11.0 Performance.

Storm's benchmark was written using the Java API. We tested both Apache Storm 0.10.0 release and a 0.11.0 snapshot.[1] One worker process per host was used, and each worker was given 16 tasks to run in 16 executors—one for each core. Storm 0.11.0 beat Storm 0.10.0, showing the optimizations that have gone recently. We omit the description of Storm 0.10.0 here due to limited space. As shown in Fig. 4, Storm 0.11.0 compared favorably to both Flink and Spark Streaming. However, at high-throughput both versions of Storm struggled.

Storm 0.11.0 performed similarly until we disabled acking. In the benchmarking topology, acking was used for flow

[1]The snapshot's Git commit hash was a8d253a.

control but not for processing guarantees. With acking enabled, 0.11.0 performed terribly at 150,000/s, which is slightly better than 0.10.0, but still worse than Spark Streaming and Flink. In 0.11.0, Storm added a simple automatic back pressure controller, allowing us to avoid the overhead of acking. With acking disabled, Storm even beat Flink for latency at high throughput. However, with acking disabled, the ability to report and handle tuple failures is also disabled.

VI. Conclusion and Future Work

It is the most interesting to compare the behaviors of these three systems. Looking at Fig. 5, we can see that Storm and Flink both respond quite linearly. This is because these two systems try to process an incoming event as it becomes available. On the other hand, the Spark Streaming system behaves in a stepwise manner, a direct result from its micro-batching design.

The throughput vs latency graph for the various systems is perhaps the most revealing, as it summarizes our findings with this benchmark. As can be see in Fig. 5 and 6, Flink and Storm have very similar performance, and Spark Streaming, while it has much higher latency, is expected to be able to handle much higher throughput by configuring its batch interval higher.

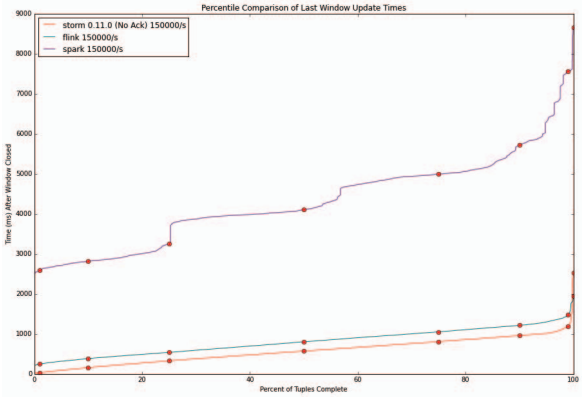


Fig. 5: Performance Comparisons of Storm, Flink and Spark Streaming in terms of window times.

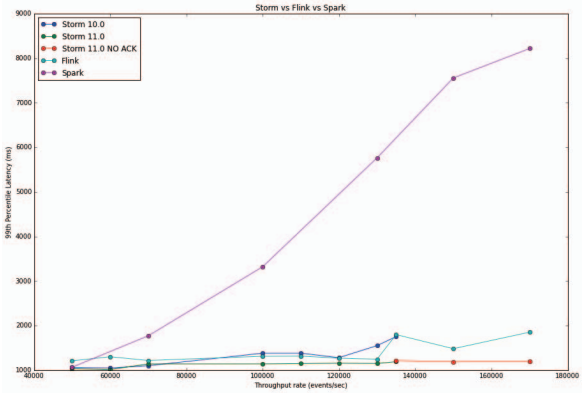


Fig. 6: Performance Comparisons of Storm, Flink and Spark Streaming in terms of 99th percentile latency.

All of these benchmarks except where otherwise noted were performed using default settings for Storm, Spark, and Flink; and we focused on writing correct, easy to understand programs without optimizing each to its full potential. We did not include the results for Storm 0.10.0 and 0.11.0 with acking enabled beyond 135,000 events per second, because they could not keep up with the throughput. The longer the topology ran, the higher the latencies got, indicating that it was losing ground.

In conclusion, Storm and Flink behave like true streaming processing systems with lower latencies, while Spark is able to handle higher throughput while having somewhat higher latencies. Some tuning is required with Spark to achieve latency SLAs. Storm's acking functionality as of version 0.11.0 incurs enough overhead to be a limitation at very high throughputs, and while processing guarantees require acking, flow control could be achieved via backpressure instead.

The competition between near real-time streaming systems is heating up, and there is no clear winner at this point. Each of the platforms studied here have their advantages and disadvantages. Performance is but one factor among others, such as security or integration with tools and libraries. Active communities for these and other big data processing projects continue to innovate and benefit from each other's advancements. We have also been working on the Storm Trident version of this streaming benchmark and look forward to expanding this benchmark for other streaming processing systems like Samza and Apex.

References

[1] "Apache Storm Project." http://storm.apache.org/.
[2] "Apache Flink Project." http://flink.apache.org/.
[3] "Apache Spark Project." http://spark.apache.org/.
[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, 2010.
[5] "Apache Samza Project." http://samza.apache.org/.
[6] "Apache Apex Project." http://apex.incubator.apache.org/.
[7] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.
[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177, IEEE, 2010.
[9] "JStorm Project." http://github.com/alibaba/jstorm/.
[10] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, ACM, 2015.
[11] "Which Stream Processing Engine: Storm, Spark, Samza, or Flink?." http://www.altaterra.net/blogpost/288668/225612/Which-Stream-Processing-Engine-Storm-Spark-Samza-or-Flink/.
[12] "High-throughput, low-latency, and exactly-once stream processing with Apache Flink.." http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/.
[13] "Streaming Big Data: Storm, Spark and Samza.." https://tsicilian.wordpress.com/2015/02/16/streaming-big-data-storm-spark-and-samza/.
[14] "Yahoo Streaming Benchmark." https://github.com/yahoo/streaming-benchmarks.
[15] "Apache Kafka Project." http://kafka.apache.org/.
[16] "Redis Project." http://redis.io/.