International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org
Volume 6, Issue 4, July- August 2017 ISSN 2278-6856

Performance Assessment of Storm and Spark for Twitter Streaming

[1]B. Revathi Reddy, [2]T.Swathi

[1]PG Student, Computer Science and Engineering Dept., GPREC, Kurnool(District), Andhra Pradesh-518004, INDIA,

[2]Assistant Professor of Computer Science and Engineering Dept., GPREC, Kurnool(District), Andhra Pradesh-518004, INDIA

Abstract
Twitter is a rich source of a user's interests: the public bio, observations, people followed, retweets and favorites. What if we could process all this information in real time to build awesome data analytics that personalize content based on the Twitter profile? Twitter 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 assessments 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 assessment for two representative computation engines: Storm and Spark Streaming. Instead of testing speed-of-light event processing, Based on our experiments, we provide a performance assessment comparison of the these two data streaming tools in terms of 99th percentile latency and throughput for various configurations.

Keywords-Twitter Streaming processing, Assessment, Storm, Spark, Low Latency

1. INTRODUCTION

Twitter's strength is real-time. No other social platform comes close on this front. While Facebook is trying to compete and Snapchat offers a unique perspective on the theme, Twitter remains our best indicator of the wider pulse of the world and what's happening within it.

This is probably the biggest tragedy of the current troubles at the mico-blog giant – if Twitter's forced to change how it operates and/or reduce its capacity to be a measure of what's happening, always and anytime, then it's not just Jack Dorsey and Twitter's shareholders that'll lose out. It's everyone. The data provided by Twitter, and the insights we're able to glean from them, can be truly world-changing, in more ways than most people realize. While the main stories you hear about the platform are Kanye West asking Mark Zuckerberg for a billion dollars or Donald Trump quoting facist dictators, there's far more to Twitter than celebrity gossip. It's a significant part, for sure, but there's much more value to be gained from tweets. And the more people use it, the higher than value becomes.

But then, of course, the opposite is also true. Twitter 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 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 Spark-2.1.0 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 assessment 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 assessment and released it as open source [14]. In this assessment, 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, And Spark), which show different advantages and shortcomings. The results demonstrate that at fairly high throughput, has 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.

2. RELATED WORK

What is Streaming Data?

Streaming Data is data that is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes). Streaming data includes a wide variety of data such as log files generated by customers using your mobile or web applications, ecommerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers.
This data needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics including correlations, aggregations, filtering, and sampling. Information derived from such analysis gives companies visibility into many aspects of their business and customer activity such as –service usage (for metering/billing), server activity, website clicks, and geo-location of devices, people, and physical goods –and enables them to respond promptly to emerging situations. For example, businesses can track changes in public sentiment on their brands and products by continuously analyzing social media streams, and respond in a timely fashion as the necessity arises.

Benefits of Streaming Data

Streaming data processing is beneficial in most scenarios where new, dynamic data is generated on a continual basis. It applies to most of the industry segments and big data use cases. Companies generally begin with simple applications such as collecting system logs and rudimentary processing like rolling min-max computations. Then, these applications evolve to more sophisticated near-real-time processing. Initially, applications may process data streams to produce simple reports, and perform simple actions in response, such as emitting alarms when key measures exceed certain thresholds.

Eventually, those applications perform more sophisticated forms of data analysis, like applying machine learning algorithms, and extract deeper insights from the data. Over time, complex, stream and event processing algorithms, like decaying time windows to find the most recent popular movies, are applied, further enriching the insights.

Streaming Data Example - Twitter
Fig 2 shows how even the Twitter streaming api is not intended to be connected by users, but by background processes that download messages in a store accessible by the HTTP. Users poll only the HTTP server, that reads the messages from the store and sends the back to the clients. As a disconnected protocol, HTTP enable massive scalability that would not be possible otherwise. If each client establishes a persistent TCP connection backed by a dedicated server thread, you will rapidly exhaust server resources! Moreover any HTTP proxy between the User Agent and the server could cause unexpected behaviors. Thus, if you are bound to the HTTP protocol, the User Agent should poll. You can reduce the network load with headers like Last-Modified/If-Modified-Since or Etag/If-None-Match. However, if you can adopt a different protocol, I strongly suggest to try a service bus over a connected TCP protocol.

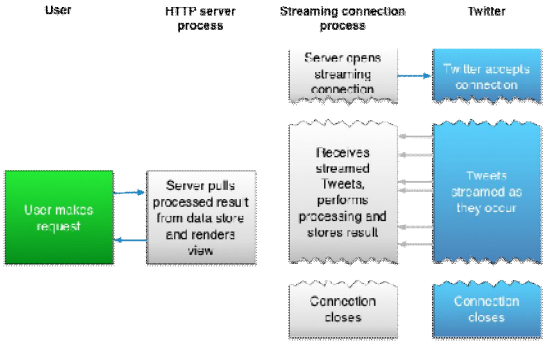

Fig. 1 Twitter API flow

3. SPARK STREAMING

For Spark assessment, 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 assessment. Storm and Spark-2.1.0assessments 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 assessment 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 assessment 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.The 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 assessment 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.

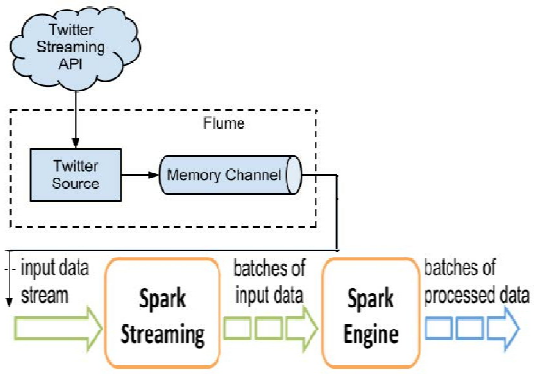

Fig 2. Spark Twitter Streaming Flow

4. STORM STREAMING

Storm architecture very much resembles to Hadoop architecture; it has two types of nodes: a master node and the worker nodes. The master node runs Nimbus that is copying the code to the cluster nodes and assigns tasks to the workers – it has a similar role as JobTracker in Hadoop. The worker nodes run the Supervisor which starts and stops worker processes – its role is similar to TaskTrackers in Hadoop. The coordination and all states between Nimbus and Supervisors are managed by Zookepeer, so the architecture looks as follows:

One of the key concepts in the Storm is topology; in essence a Storm cluster executes a topology – topology defines the data sources, the processing tasks and the data flow between the nodes. Topology and MapReduce jobs in Hadoop can be considered analogous.

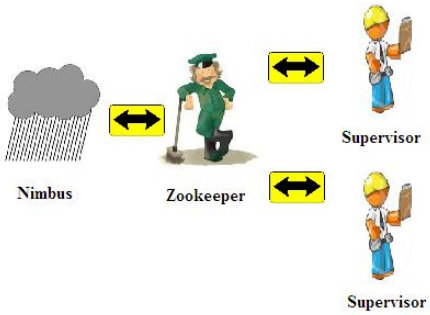

Fig.3: Storm Architecture flow

Storm has a concept of streams which are basically a sequence of tuples, they represent the data that is being passed around the Storm nodes. There are two main components to manipulate stream data: spouts which are reading data from a source (e.g. a queue or an API, etc) and emit a list of fields. Bolts are consuming the data coming from input streams, processing them and then emit a new stream or store the data in a database.

One important thing when you define a topology is determine how data will be passed around the nodes. As discussed above, a node (running either spouts our bolts) will emit a stream. Stream grouping functionality will allow to decide which node(s) will receive the emitted tuples. Storms has a number of grouping functions like shuffle grouping (sending streams to a randomly chosen bolt), fields grouping (it guarantees that a given set of fields is always sent to the same bolt), all grouping (the tuples are sent to all instances of the same bolt), direct grouping (the source determines which bolt receives the tuples) and you can implement your own custom grouping method, too.

5. ASSESSMENTMODEL

Our streaming assessment 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 assessment 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):

- Read an event from Kafka.
- Deserialize the JSON string.
- Filter out irrelevant events (based on event type field)
- Take a projection of the relevant fields (ad_id and event time)
- 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:

[1] user_id, page_id, ad_id: UUID

[2] ad_type: String in banner, modal, sponsored-search, mail, mobile

TABLE I. event type: String in view, click, purchase

TABLE II. event time: Timestamp

TABLE III. 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 Spark-2.1.0, 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 u pdated at*. After each run, a utility reads windowsfrom Redis and compares the windows' start times to their *last u pdated at* times, yielding a latency data point. Wecalculate the event latency by subtracting the window start time and duration time from the *last u pdated at* time.

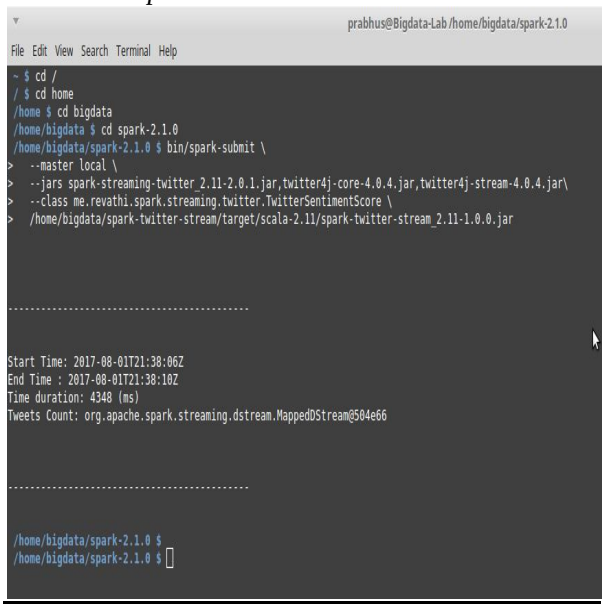

Fig.4 Spark Output

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 assessment. 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.

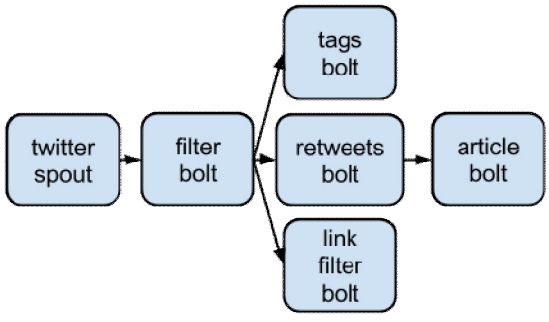

Fig.5: Storm Twitter streaming

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 assessment. 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 assessment we run 100 campaigns, with 10 ads per campaign.

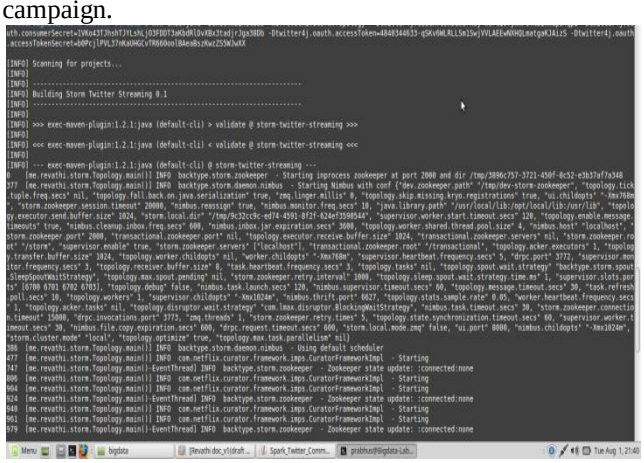

Fig.6 Storm output

At the beginning, the assessment'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 assessment utility was then run to generate a file containing a list of *last u pdated at* −*start time* numbers. These files were savedfor each throughput we tested and then were used to generate the charts in this document. This assessment is implemented in three different versions, respectively for Spark-2.1.0, Spark and Storm. In Section III, IV and V, we will describe each implementation and its results in details.

The final results are interesting. There are essentially three
Storm's assessment was written using the Java API. We tested both Apache Storm 0.10.0 release and a 0.11.0 snap shot.[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 1.1.0 beat Storm 0.10.0, showing the optimizations that have gone recently.
.
Storm 1.1.0 performed similarly until we disabled acking. In the assessmenting topology, acking was used for flow 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 Spark-2.1.0. 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 Spark for latency at high throughput. However, with acking disabled, the ability to report and handle tuple failures is also disabled.

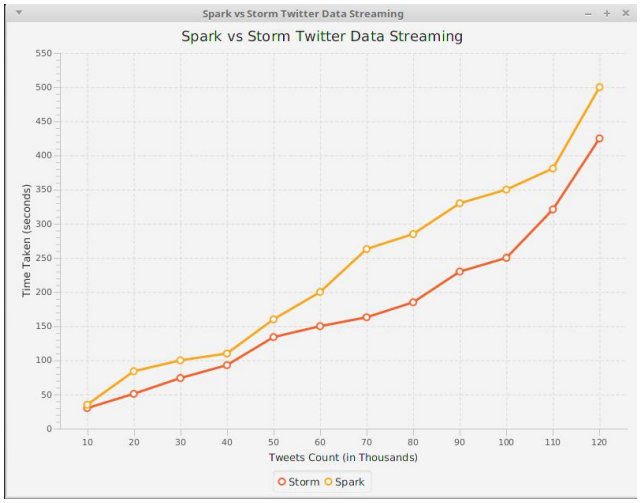

Fig.7 Spark and Storm Streaming Performance

6. 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 Spark 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 assessment. As can be see in Fig. 5 and 6, Spark 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.

All of these assessments except where otherwise noted were performed using default settings for Storm, Spark, andSpark-2.1.0; 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 Spark-2.1.0 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 assessment and look forward to expanding this assessment for other streaming processing systems like Samza and Apex.

References

[1] "Apache Storm Project." http://storm.apache.org/.
[2] "Apache Spark-2.1.0 Project." http://Spark-2.1.0.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. Fernandez´-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 process-ing" process-ing," 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), 2010IEEE 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 In-ternational Conference on Management of Data, pp. 239–250, ACM,2015.
[11] "Which Stream Processing Engine: Storm, Spark, Samza, or Spark-2.1.0?." http://www.altaterra.net/blogpost/288668/225612/Which-Stream-Processing-Engine-Storm-Spark-Samza-or-Spark-2.1.0/.
[12] "High-throughput, low-latency, and exactly-once stream processing with Apache Spark-2.1.0.." http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-Spark-2.1.0/.
[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 Assessment." https://github.com/yahoo/streaming-assessments.
[15] "Apache Kafka Project." http://kafka.apache.org/.
[16] "Redis Project." http://redis.io/

Author

B.Revathi, is a P.G Scholar in CSE Dept at G.Pulla Reddy Engineering college (Autonomus) Kurnool (district), A.P.518002, India.

T.Swathi M.Tech[Ph.D], working as Assistant Professor in CSE Dept, G.Pulla Reddy Engineering college (Autonomus)Kurnool (district), A.P. 518002, India. Her reasearch interests are in the area of cloud computing & Big Data.