Benchmarking Streaming Computation Engines at Yahoo!

(Yahoo Storm Team in alphabetical order) Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng and Paul Poulosky.

will rerun and repost the numbers when we have fixed this.

UPDATE: Dec 18, 2015 there was a miscommunication and the code that was checked in was not the exact code we ran with for flink. The real code had the debugging removed. Data-Artisans has looked at the code and confirmed it and the current numbers are good. We will still rerun at some point soon.

Executive Summary - Due to a lack of real-world streaming benchmarks, we developed one to compare Apache Flink, Apache Storm and Apache Spark Streaming. Storm 0.10.0, 0.11.0-SNAPSHOT and Flink 0.10.1 show sub- second latencies at relatively high throughputs with Storm having the lowest 99th percentile latency. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.

At Yahoo, we have invested heavily in a number of open source big data platforms that we use daily to support our business. For streaming workloads, our platform of choice has been Apache Storm, which replaced our internally developed S4 platform. We have been using Storm extensively, and the number of nodes running Storm at Yahoo has now reached about 2,300 (and is still growing).

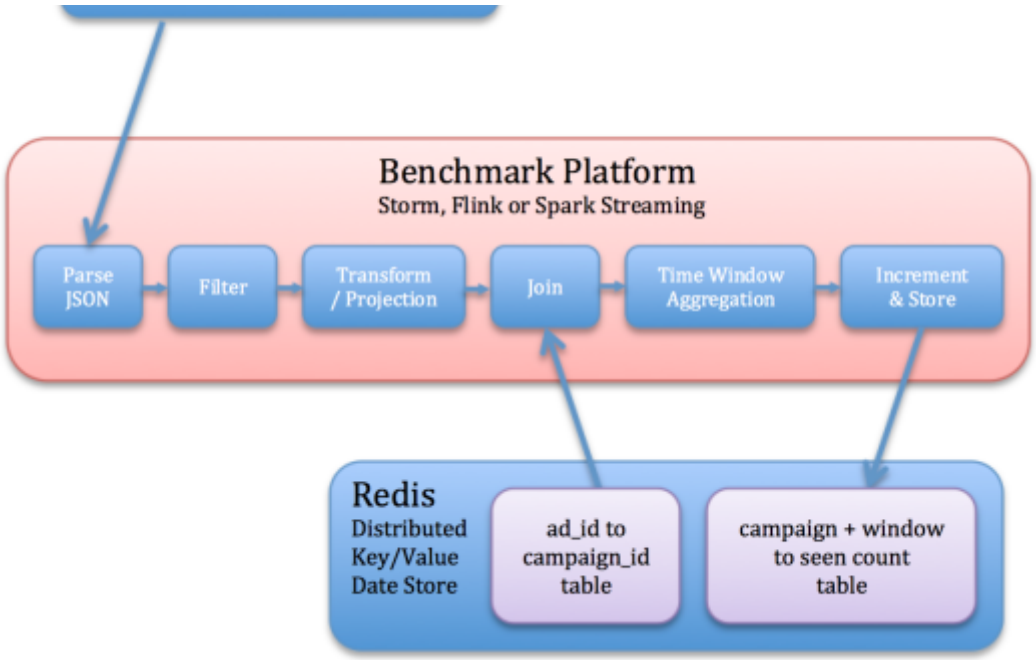
Since our initial decision to use Storm in 2012, the streaming landscape has changed drastically. There are now several other noteworthy competitors including Apache Flink, Apache Spark (Spark Streaming), Apache Samza, Apache Apex and Google Cloud Dataflow. There is increasing confusion over which package offers the best set of features and which one performs better under which conditions (for instance see here, here, here, and here).

To provide the best streaming tools to our internal customers, we wanted to know what Storm is good at and where it needs to be improved compared to other systems. To do this we started to look for stream processing benchmarks that we could use to do this evaluation, but all of them were lacking in several fundamental areas. Primarily, they did not test with anything close to a real world use case. So we decided to write one and released it as open source https://github.com/yahoo/streaming-benchmarks. In our initial evaluation we decided to limit our test to three of the most popular and promising platforms (Storm, Flink and Spark), but welcome contributions for other systems, and to expand the scope of the benchmark.

The benchmark is a simple advertisement application. 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 (and shown in the following figure):

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*: UUID
- *page_id*: UUID
- *ad_id*: UUID
- *ad_type*: String in {banner, modal, sponsored-search, mail, mobile}
- *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

last updated.

After each run, a utility reads windows from Redis and compares the windows' times to their *last_updated_at* times, yielding a latency data point. Because the last event for a window cannot have been emitted after the window closed but will be very shortly before, the difference between a window's time and its last_updated_at time minus its duration represents the time it took for the final tuple in a window to go from Kafka to Redis through the application.

window.final_event_latency = (window.last_updated_at – window.timestamp) – window.duration

This is a bit rough, but this benchmark was not purposed to get fine-grained numbers on these engines, but to provide a more high-level view of their behavior.

Benchmark setup

- 10 second windows
- 1 second SLA
- 100 campaigns
- 10 ads per campaign
- 5 Kafka nodes with 5 partitions
- 1 Redis node
- 10 worker nodes (not including coordination nodes like Storm's Nimbus)
- 5-10 Kafka producer nodes
- 3 ZooKeeper nodes

Since the Redis node in our architecture only performs in-memory lookups using a well-optimized hashing scheme, it did not become a bottleneck. The nodes are homogeneously configured, each with two Intel E5530 processors running at 2.4GHz, with a total of 16 cores (8 physical, 16 hyperthreading) per node. Each node has 24GiB 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.

benchmark.

The use of 10 workers for a topology is near the average number we see being used by topologies internal to Yahoo. Of course, our Storm clusters are larger in size, but they are multi-tenant and run many topologies.

To begin the benchmarks 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 *window.last_updated_at – window.timestamp* numbers. These files were saved for each throughput we tested and then were used to generate the charts in this document.

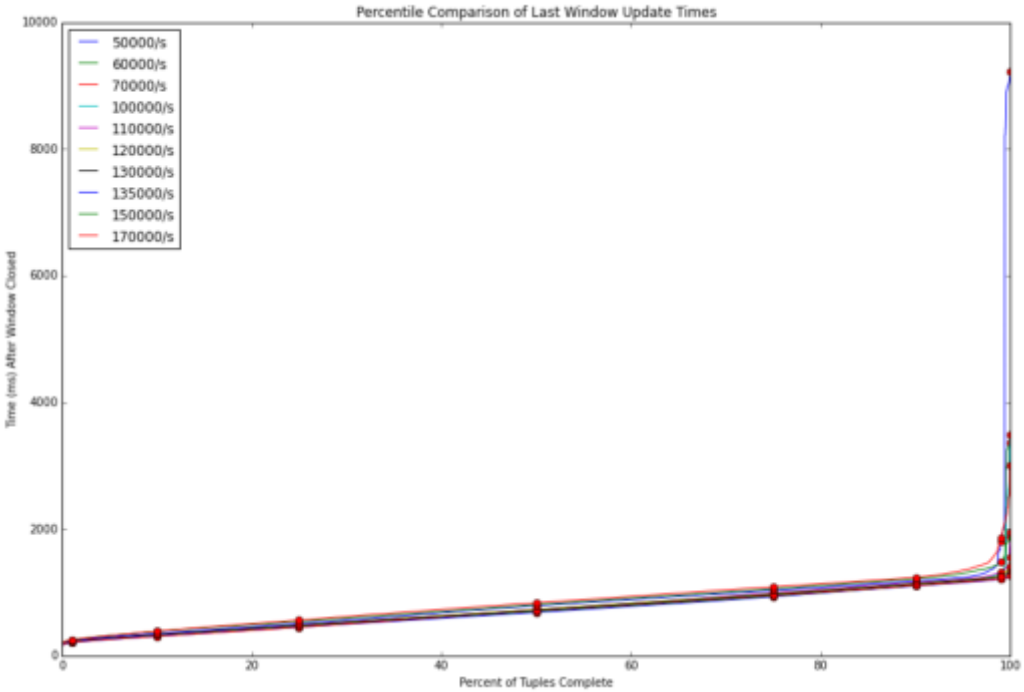
Flink

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 configs we used in Flink is listed below:

- taskmanager.heap.mb: 15360
- taskmanager.numberOfTaskSlots: 16

The Flink version of the benchmark uses the FlinkKafkaConsumer to read data in from Kafka. The data read in from Kafka—which is in a JSON formatted string— 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

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 graph below.



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.

Spark

For the Spark benchmark, the code was written in Scala. Since the micro-batching methodology of Spark 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

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. Note that in the subsequent discussion, we use the term “RDD” instead of “DStream” to refer to the RDD representation of the DStream in the currently active microbatch. Processing begins with the direct Kafka consumer included with Spark 1.5. Since the Kafka input data in our benchmark is stored in 5 partitions, this Kafka consumer creates a DStream with 5 partitions as well. After that, a number of transformations are applied on the DStreams, including maps and filters. The transformation involving joining data with Redis is a special case. Since we do not want to create a separate connection to Redis for each record, we use a mapPartitions operation that can give control of a whole RDD partition to our code. This way, we create one connection to Redis and use this single connection to query information from Redis for all the events in that RDD partition. The same approach is used later when we update the final results in Redis.

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.

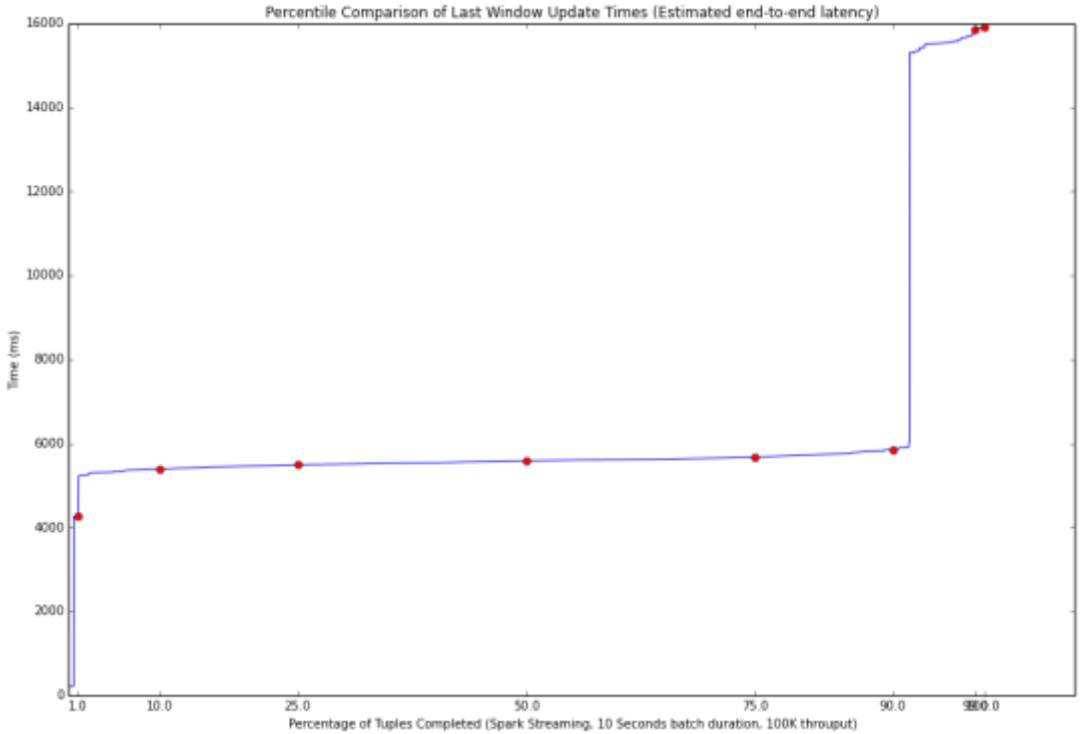
We found that with high enough throughput, Spark was not able to keep up. At 100,000 messages per second the latency greatly increased. We considered adjustments along two control dimensions to help Spark cope with increasing throughput.

The first is the microbatch duration. This is a control dimension that is not present in a pure streaming system like Storm. Increasing the duration increases latency while reducing overhead and therefore increasing maximum throughput. The challenge is that the choice of the optimal batch duration that minimizes latency while allowing spark to handle the throughput is a time consuming process. Essentially, we have to set a batch duration, run the benchmark for 30 minutes, check the results and decrease/increase the duration.

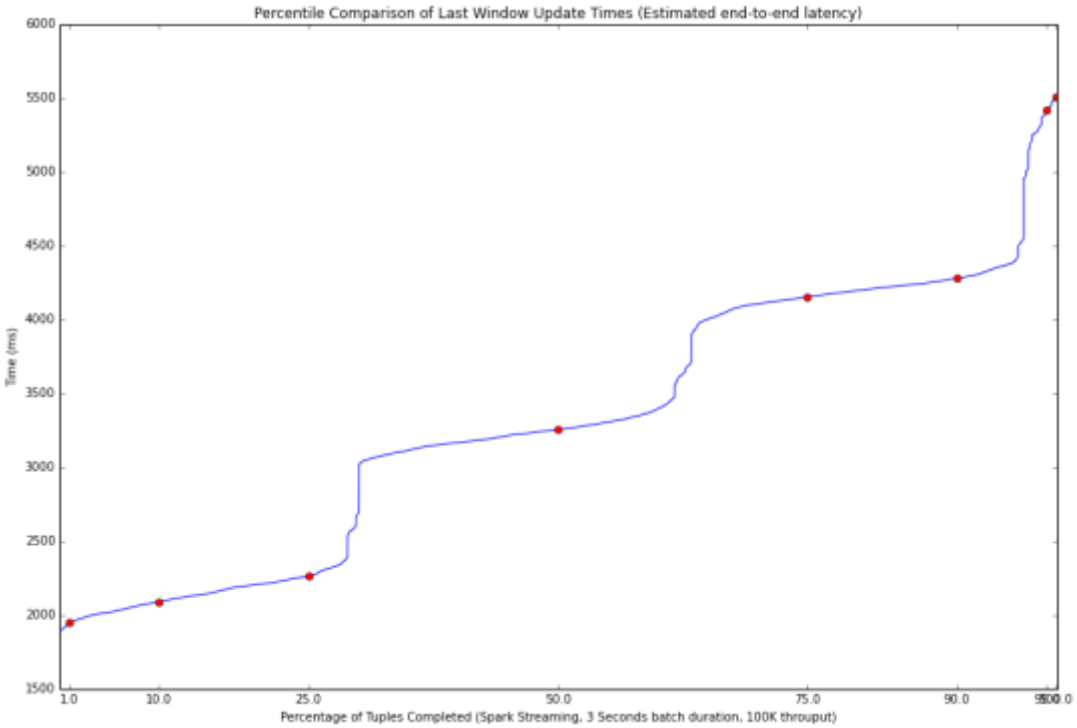
The second dimension is parallelism. However, increasing parallelism is simpler said than done in the case of Spark. For a true streaming system like Storm, one bolt instance can send its results to any number of subsequent bolt instances by using a random shuffle. To scale, one can increase the parallelism of the second bolt. In the case of a micro batch system like Spark, we need to perform a reshuffle operation similar to how

found the bottleneck to be scheduling, and so reshuffling only added overhead. We suspect that at higher throughput rates or with operations that are CPU-bound, the reverse would be true.

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. The following figure 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



Finally, the third behavior is when Spark streaming falls behind. In this case, the benchmark takes a few minutes after the input data finishes to process all of the events. This situation is shown in the following figure. Under this undesirable operating region, Spark spills lots of data onto disks, and in extreme cases we could end up running out of disk space.

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. The third operating region is where back pressure shows the most negative impact. It changes the batch length, but Spark still cannot cope with the throughput and falls behind. This is shown in the next figures. Our experiments showed that the current back pressure implementation did not help our benchmark, and as a result we disabled it.

YAHOO!

ARCHIVE

ENGINEERING

Engineering

Research

Hadoop

Cloud Services

Open Source

Node @ Yahoo

OTHER Y! BLOGS

Yahoo

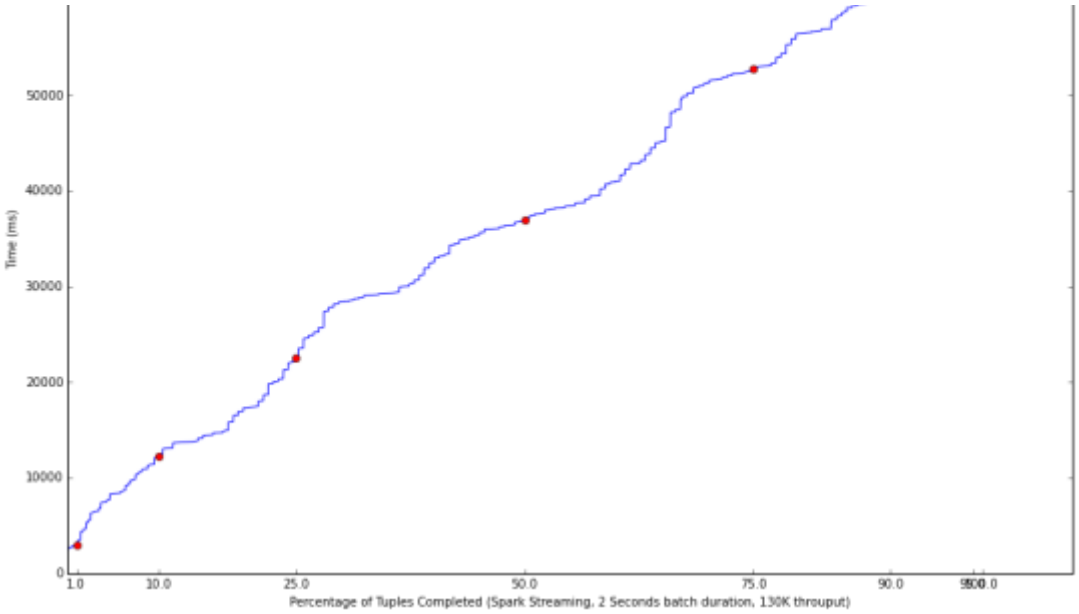
Search

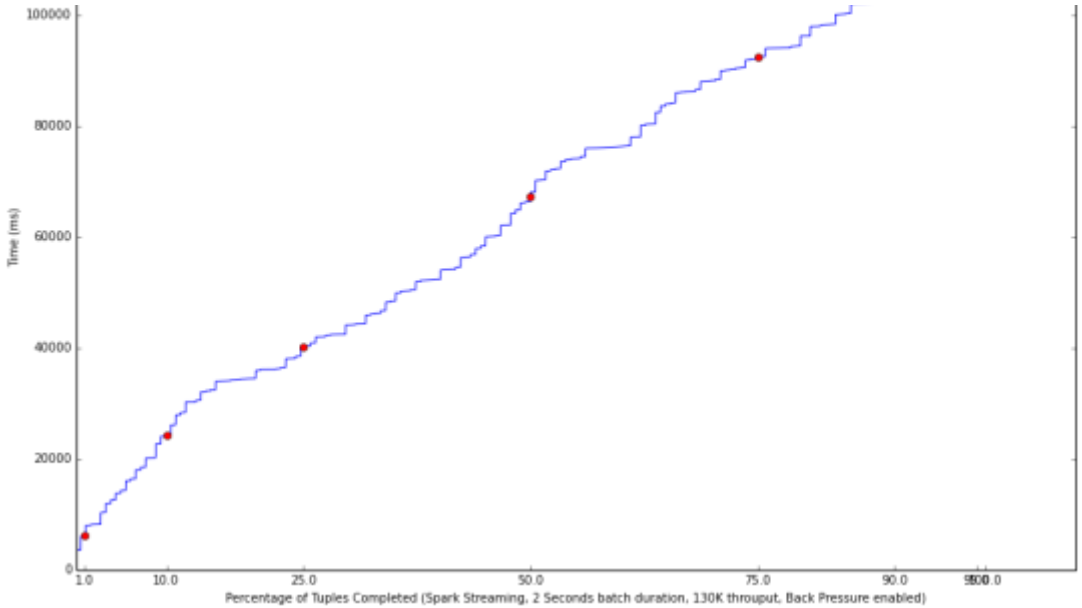
Messenger

Mail

Sports

Answers





Performance without back pressure (top), and with back pressure enabled (bottom). The latencies with the back pressure enabled are worse (70 seconds vs 120 seconds). Note that both of these results are unacceptable for a streaming system as both fall behind the incoming data. Batch duration was set to 2 seconds for each run, with 130,000 throughput.

Storm

Storm's benchmark was written using the Java API. We tested both Apache Storm 0.10.0 release and a 0.11.0 snapshot. The snapshot's commit hash was a8d253a. One worker process per host was used, and each worker was given 16 tasks to run in 16 executors - one for each core.

Storm 0.10.0:

YAHOO!

ARCHIVE

ENGINEERING

Engineering

Research

Hadoop

Cloud Services

Open Source

Node @ Yahoo

OTHER Y! BLOGS

Yahoo

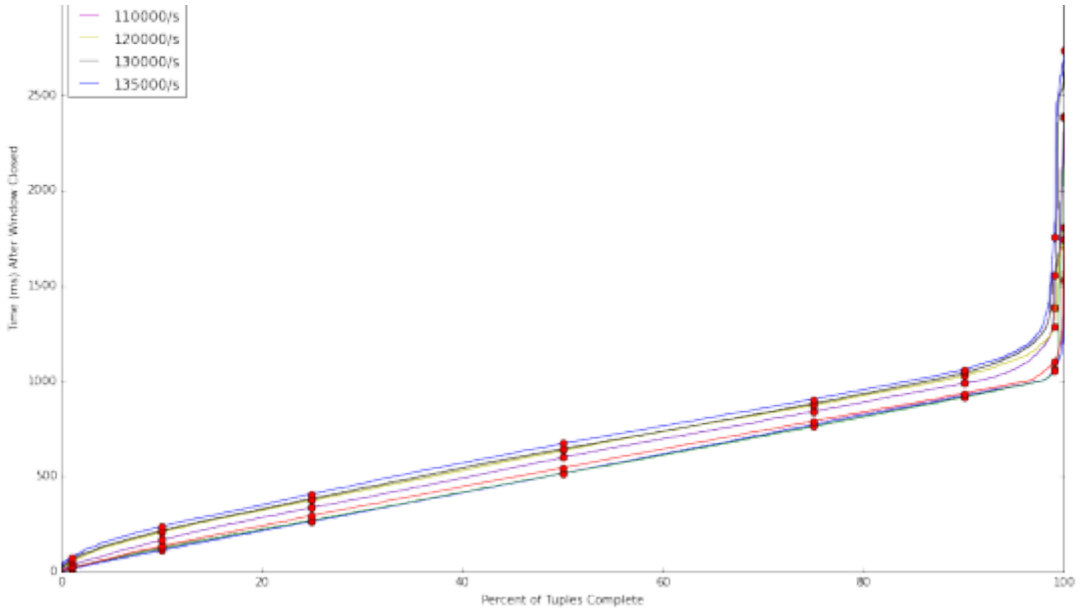
Search

Messenger

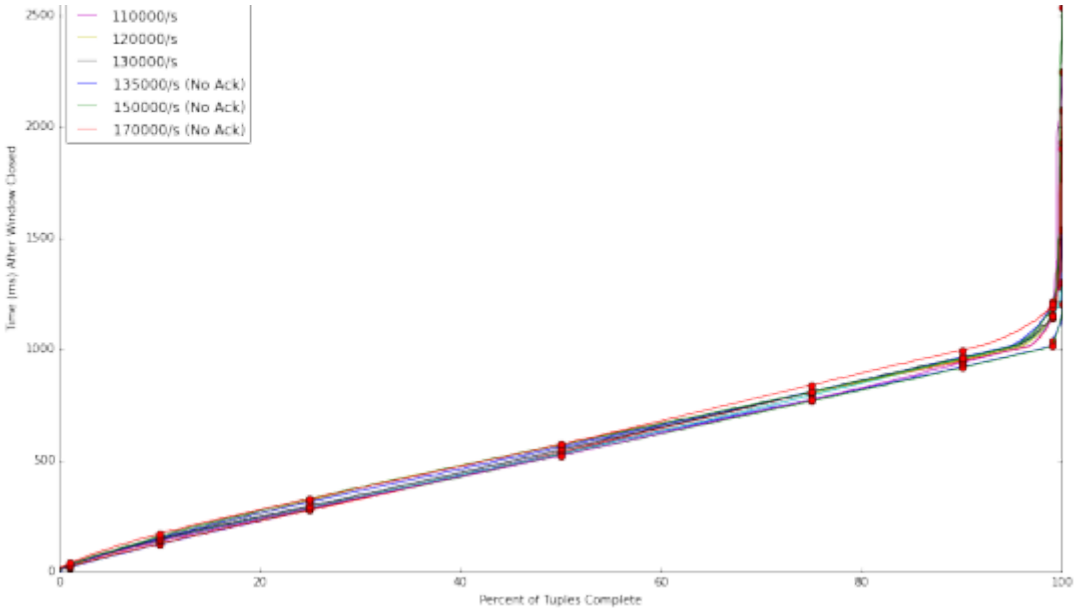
Mail

Sports

Answers



Storm 0.11.0:



Storm compared favorably to both Flink and Spark Streaming. Storm 0.11.0 beat Storm 0.10.0, showing the optimizations that have gone in since the 0.10.0 release. However, at high-throughput both versions of Storm struggled. Storm 0.10.0 was not able to handle throughputs above 135,000 events per second.

Storm 0.11.0 performed similarly until we disabled acking. In the benchmarking topology, acking was used for flow control but not for processing guarantees. In 0.11.0, Storm added a simple back pressure controller, allowing us to avoid the overhead of acking. With acking enabled, 0.11.0 performed terribly at 150,000/s—slightly better than 0.10.0, but still far worse than anything else. 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 disabled also.

Conclusions and Future Work

It is interesting to compare the behavior of these three systems. Looking at the following figure, we can see that Storm and Flink both respond quite linearly. This is because these two systems try to process an incoming event

YAHOO!

ARCHIVE

ENGINEERING

Engineering

Research

Hadoop

Cloud Services

Open Source

Node @ Yahoo

OTHER Y! BLOGS

Yahoo

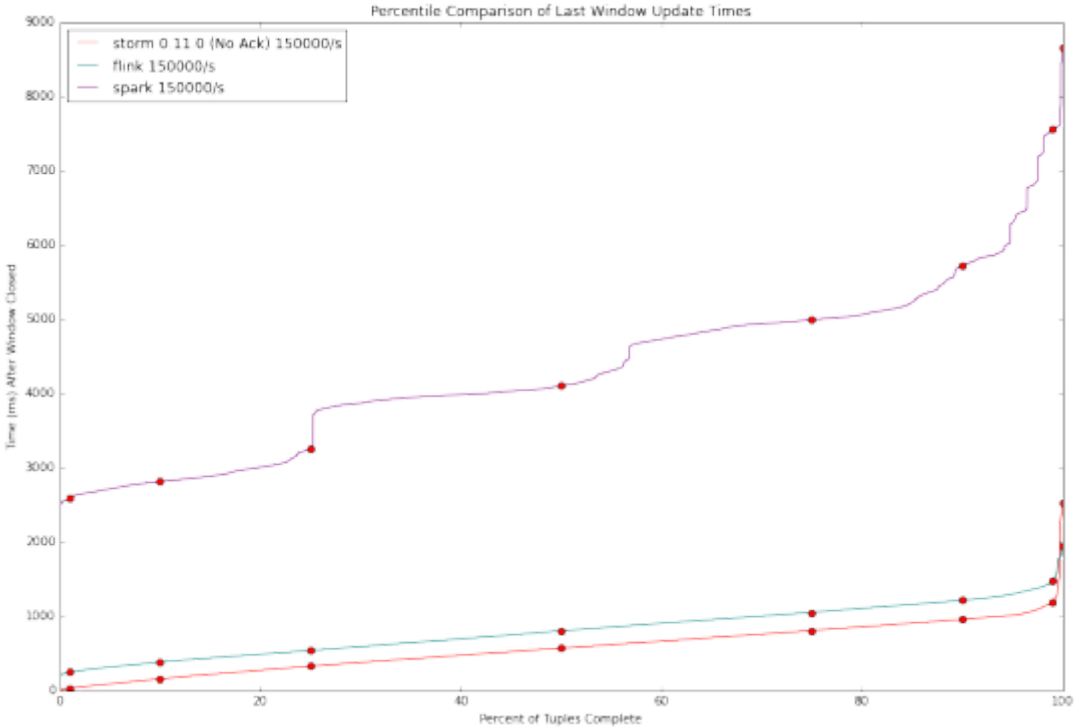
Search

Messenger

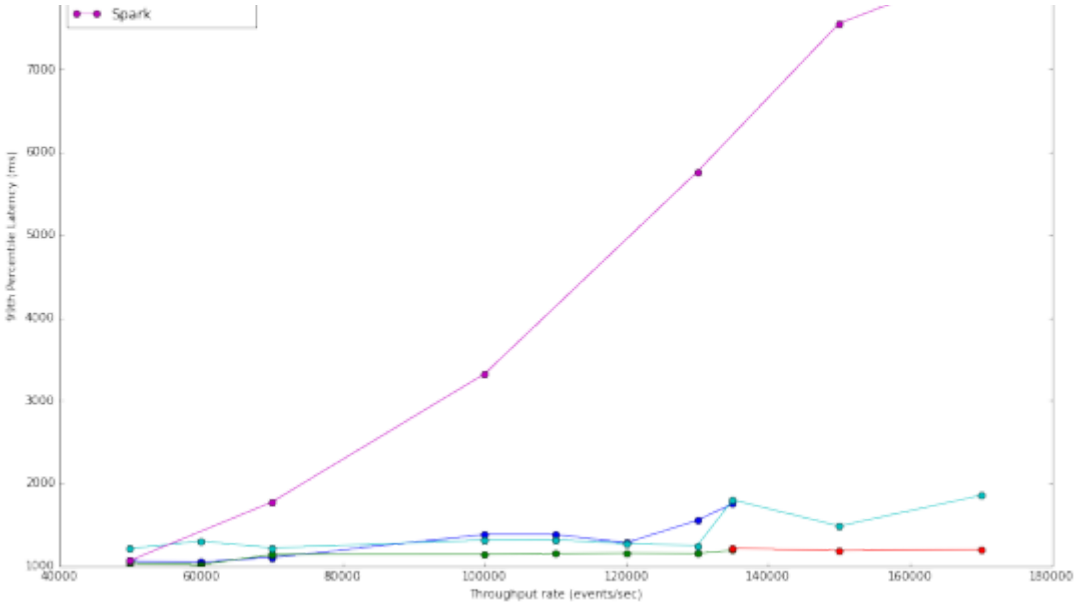
Mail

Sports

Answers



The throughput vs latency graph for the various systems is maybe the most revealing, as it summarizes our findings with this benchmark. 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.



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 resulting graph had the final point for Storm 0.10.0 in the 45,000 ms range, dwarfing every other line on the graph. The longer the topology ran, the higher the latencies got, indicating that it was losing ground.

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. Because of this each of the six steps were a separate bolt or spout. Flink and Spark both do operator combining automatically, but Storm (without Trident) does not. What this means for Storm is that events go through many more steps and have a higher overhead compared to the other systems.

In addition to further optimizations to Storm, we would like to expand the benchmark in terms of functionality, and to include other stream processing systems like Samza and Apex. We would also like to take into account fault tolerance, processing guarantees, and resource utilization.

YAHOO!

ARCHIVE

ENGINEERING
Engineering
Research
Hadoop
Cloud Services
Open Source
Node @ Yahoo

OTHER Y! BLOGS
Yahoo
Search
Messenger
Mail
Sports
Answers

acking, Storm with acking enabled would compete with Flink at very high throughput too.

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 look forward to expanding this benchmark and testing newer releases of these systems as they come out.

apachestorm *apachespark* *apacheflink* *streaming* *realtime* *benchmark* *yahoo* *performance* *open source* *yahoo engineering*

By ***revans2***

DEC 16TH, 2015 37

SHARE

G+1 13 Pin it

NOTES

semanticearth-community reblogged this from yahooeng

super-aironman liked this

arganoilhome liked this

yagsachin liked this

nicola-barbieri liked this

YAHOO!

ARCHIVE

ENGINEERING

Engineering

Research

Hadoop

Cloud Services

Open Source

Node @ Yahoo

OTHER Y! BLOGS

Yahoo

Search

Messenger

Mail

Sports

Answers

muraliparimi liked this

mjuarez liked this

pythoneer007 liked this

zlckr liked this

lonelybob liked this

its-futuristiccollectorreview liked this

paulobsf liked this

joeinjesus liked this

tommychheng liked this

devenpanchal reblogged this from yahooeng

pkvprakash liked this

beitao liked this

rodrigosqsz liked this

yahoohadoop reblogged this from yahooeng

mrugen liked this

sarpunk liked this

amotzm reblogged this from yahooeng

benblack86 liked this

geekflght reblogged this from yahooeng

geekflght liked this

revans2 liked this

anujbahuguna liked this