**Project Report**

CS764 Fall 2016

# Spark v/s Heron: Comparison of Stream Processing Systems

Rohit Damkondwar (damkondwar@wisc.edu)

Pradeep Kashyap Ramaswamy (pradeep.kashyap@wisc.edu)

## Abstract

There has been an explosion of data in recent past and so does the requirements to process them. Stream processing in one of the programming paradigms that are used to digest the data and garner meaningful information from it. The difference from regular data processing systems is that, Stream processing systems aren't programs which are run once on a set of input and produce output, but are programs which always run and process live data. Twitter, Facebook, and other data-centric companies have built many of their products using these Streaming System. Our aim is to compare two such Stream Processing systems: Spark and Heron. We have come up with two workloads which are run on these systems with various configurations of these Systems. We have presented the results of these workloads in this report and have discussed the results in detail in the following sections.

## 1. Introduction.

The motivation to this project arises from the fact there are no existing results of comparison of these two open-source systems—Spark and Heron—for their performance. We have built a benchmarking test suite for the same purpose and the details are discussed in this report. Spark [1] was initially developed at UC Berkeley in 2009 and has been evolving through open-source community support. Heron [2] was developed at twitter to supersede storm to meet their scale of streaming processing requirements. Both streaming systems have been employed by start-ups to well-known companies. As heron is relatively new, we believe that our benchmarking to compare Spark and Heron would give a better insight on streaming capabilities of them.

The rest of the report is organized in the following way: **Section 2** discusses the goal of the project explaining problem statement and how our work is related to it. **Section 3** describes about our approach of comparing the systems (from here-on, '*systems*' meaning Spark and Heron) and the workloads we have chosen. It also includes the test environment that we've used for running the benchmarks including the software package and tools. **Section 4** discusses the results of the benchmarking tests and our analysis of the results. **Section 5** gives a summary and conclusion to the report highlighting important takeaways from this report.

## 2. Goal of the Project.

As explained before, our focus lies in comparing the performance of Spark and Heron on various use-cases. Our work provides performance insights on these systems and it is highly relevant because there exists no other precedent work that compares the performance of Spark and Heron. We take a pragmatic approach in comparing these two systems by testing them again relevant workloads (refer table 1). Our goal is to collected the results of these tests and provided our analysis and insights into the performance and behaviors of these two system. The types of workloads and their characteristics are discussed in the next section.

| Workload | Receiver Heavy | Processor Heavy | Comments | Implemented ? |
|---|---|---|---|---|
| **Hashtag Counter** | Yes | No. Very light | Possible Workload where more time is spent in reading tuples. | ✓ |
| **Perceptron** | Yes | Yes | Regular workload where time is spent on both receiving and processing tuples | ✓ |
| **NULL** | No | No. | No work in either stages. Not practical | ✗ |
| **Processor Intensive only** | No | Yes | This is more like a case where there is no input and only processing. Not practical | ✗ |

Table 1: All possible workloads and our implementations.

## 3. Our Proposed approach.

In any Stream processing workloads, there are two important phases: Receiving and the Processing. The *Receiver* task does the fetching part of the data to be processes and the *Processor* task does the processing part of it. For our benchmarking suite, we have chosen Hashtag Count and Perceptron (Table 1) as the workloads. Hashtag Counter, as the name indicates, does the frequency count of the hashtags. This is a real-life use case in Twitter where counting the number of times a #hashtag is used to predict the trending topics. This use-case is more of receiver intensive than processor; The receiver fetches the tuples and feeds it to the processor and the processor uses a Hashmap to count the frequency of the hashtag. *Perceptron* [8] is a Machine Learning algorithm for supervised learning of binary classifiers. This is a linear classifier algorithm that predicts input based on predictor function combining a set of weights with feature vector [9]. This workload is a both Receiver and Processor intensive use-case. We have decided that these two test cases—Hashtag Count and Perceptron—would suffice for comparison as they include the practical set of use-cases in which Streaming systems are employed.

We have decided not to implement the thirds and fourth test (from Table 1) cases as they seem impractical; There exists no use case of employing any system without input and processing requirements (NULL). The fourth test case is also not practical as there is no input to the system and it doesn't make sense with process intensive workload without any input. This can only be used to test the working and behavior of the program (Internally generated tuples), thus for our benchmark, we have not included this test case.

We have implemented word frequency count of hashtags on twitter tweets. The **twitter dataset** used here is available at [3]. Apache Kafka [4] will be used as input to both the systems. We load these hashtags into Kafka topic. Both systems monitor Kafka topic for stream of tweets and calculate count (frequency) of each hashtag till the end of stream. We record the time required for each system to consume variable number of tweets (in the order of millions). We can then easily calculate throughput of each system. The workload Word Frequency Count is not expensive operation and hence is a good way to benchmark performance of the systems. Similarly, for Perceptron, we have fed the tuples into the Kafka. The Perceptron algorithm uses a pre-trained (training is not part of the streaming) model to predict the output for the tuple.

## 3.1 Test Environment

For our test, we have used the cloud lab's cluster with 6 nodes: one input node, one master node (submission of topologies/Scheduler) and 4 worker nodes for Computations. Each of these computers run a separate Kafka service from which the tuples are read. To make sure that Kafka is ready with the data, we feed the Kafka topics with all the input data and only after Kafka is done reading the data, we start the programs on both the systems. Spark and Heron runs on their latest versions [5] of 2.0.1 and 0.14.5 respectively.

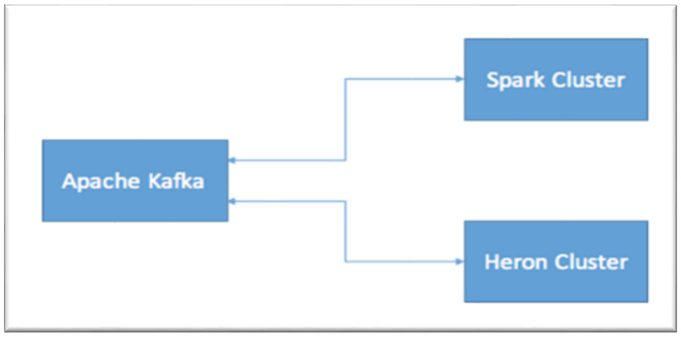

Figure 1. Heron and Spark Interaction with Kafka.

In our test, Kafka is the only source of input to both Heron and Spark clusters. As you can see from the Figure 2, Apache Kafka is installed on control node which feeds the Streaming Systems. All the worker nodes communicate over the network to fetch the tuples from Kafka. For comparison purposes (speed-up), we have two configurations of Kafka: With partition and without Partition. Partitions enables us to parallelly read tuples from the Kafka. Without partitions, Kafka has only one thread which serves the requests—used for testing Spark and Heron without parallelism; i.e., Single sequence of operations. Though it might seem that Kafka might be a bottle neck, Kafka produces lot faster than either Heron Spout or Spark can consume. With partitions in Kafka, multiple spouts in Heron can parallelly consume tuples into the system (and probably send it parallelly to bolts as well and in Spark, DStream are populated in parallel with partitions corresponding to each Kafka Partition.

### 3.1.1 Spark Cluster Setup

As observed from Figure 2, CP-1 is the spark master node which takes care of scheduling jobs across the Spark slaves—which are CP-2 to CP-5 through HDFS. Scheduler dynamically determines one of the workers as driver to execute the Spark Job. To achieve ***at least once*** data semantics, we have used HDFS for reliable meta-data checkpointing. CP-1 runs NameNode and CP-2 to CP-5 run data nodes. We have used Spark's native Standalone cluster manager

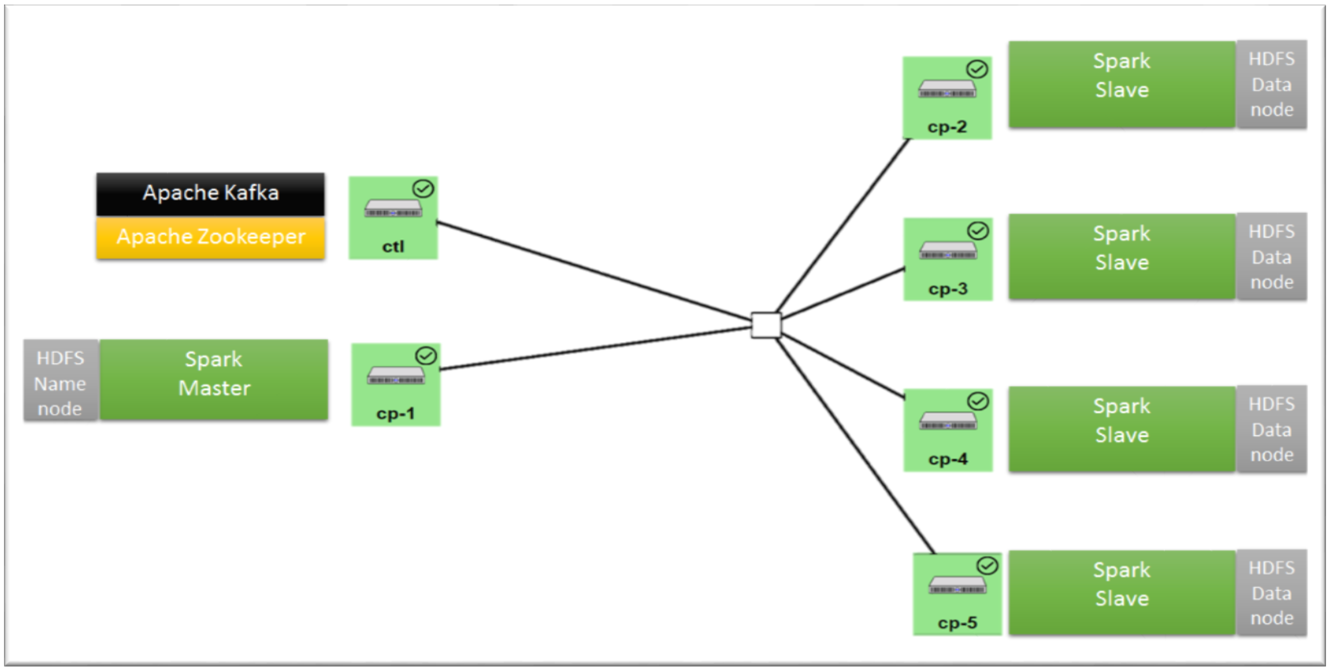

Figure 2: Spark Cluster Setup

### 3.1.2 Heron Cluster Setup

Heron's setup is similar to Spark; CP2-CP5 are worker nodes and CP-1 is the master node where heron in installed and Topologies are submitted. Every node in the cluster is installed with Mesos framework for running the heron jobs. On top of this, Aurora scheduler is used to schedule Heron Topologies. There are two Zookeepers in the Cluster: One is the Heron's Zookeeper where the meta data—user of the launched job, time of the job, and execution details—of the Topologies are kept, Second Zookeeper is used for Apache Kafka which keeps the meta data about Kafka.

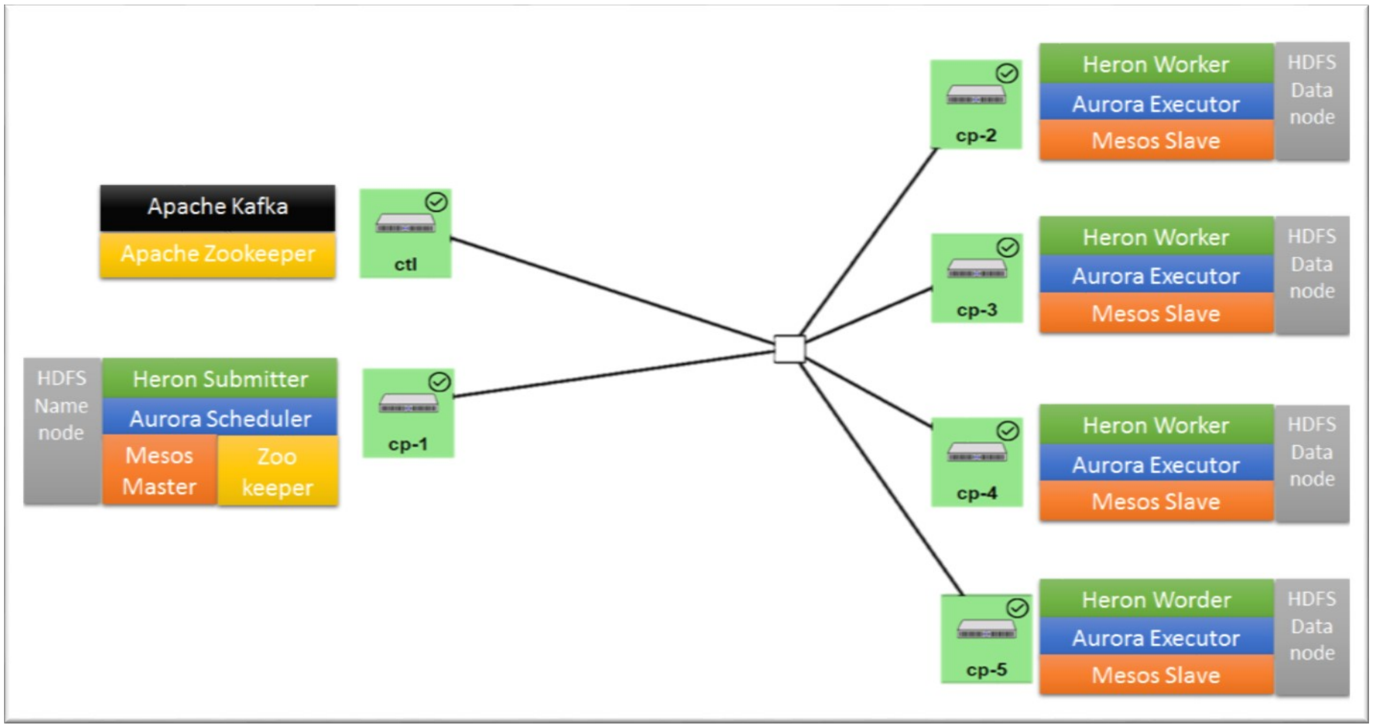

Figure 3: Heron Cluster Setup.

## 3.2 Workloads

In this section, we would discuss the two workloads which we choose earlier in this section: Hashtag Counts and Perceptron. We'll explain how these workloads are implemented in Spark and Heron with various configurations. Both workloads are tested with and without parallelism; This helps us understand how much Spark and Heron are speeding-up (scaling comparison) with the parallelism introduced. As spark does micro-batching, various bathing intervals are chosen: 50MS, 100 MS, 200ms, 500ms, 1000ms and 2000ms. Spark exhibits At-least-once semantics with Kafka Direct Streams and HDFS metadata check pointing. With heron, there are two configurations: ACK enabled and ACK disabled to compare the performance of heron for *At-most-once* and *At-least-once* semantics.

### 3.2.1 Hashtag Count:

This test case counts the frequency of words—a real world case would be to count the hashtags in tweets (Hashtag dataset is taken from [3]). In this test, the tuples will be hashtags separated by newline. To keep things consistent between Heron and Spark, we are feeding these tuples to Apache *Kafka* and then reading it from Kafka. For verification of correctness of behavior of our programs, we dumped the < hashtag, count > pair and sorted them on their name and then frequency. We used *diff* and *comm* commands to verify the outputs. The number of hashtags is about **25 million.**

**Spark (Without Parallelism):** For setup, we spawned one master and one worker. For polling Kafka, we have used Kafka integration with Spark. The Spark program creates direct stream with Kafka, polls for the stream of tweets, creates DStream object and starts processing. At the end of each batch, the spark program updates a global state map to store frequency of each hashtag from the start of the program. DStreams are extension of RDD (Resilient Distributed Datasets) [1] for streaming purposes.

**Spark (With Parallelism):** In this configuration, we spawned one master, four workers with four partitions in Kafka. The Spark program creates a direct stream with 4 threads to read parallelly from Kafka. Each thread reads from one Kafka partition. Other details are similar to Spark without Parallelism.

**Heron (Without Parallelism)**: We have one Spout and one Bolt. The Spout emits a tuple and the bolt counts the tuple (Hashtags) frequency using an in-memory map. We have interfaced our Systems with Kafka using Kafka Spouts. The Kafka Spout emits the tuples and are processed by bolts to populate the map. For Measurement for throughput, the bolt keeps track of number of tuples that it has processed. For each one million or so tuples processed, it prints the amount of time (in ms) it took to process the all tuples from the beginning. This helps in accounting the relative and absolute time taken for processing the tuples.

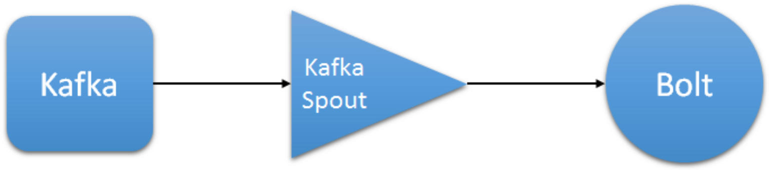

Figure 4: Heron Hashtag Count Topology without Parallelism.

**Heron (With Parallelism)**: With cluster environment, we can have four Spouts and one Bolt. Here there cannot be multiple bolts because, finally, only one bolt has the aggregate the counts. With this configuration, we use 4 partitions in Kafka so that all 4 Kafka spouts can read parallelly. All four spouts then emit the tuples into the topology and single bolt aggregates counts of these tuples. Everything else remains same as previous environment. In this test case, we cannot get the speed-up because single aggregator capacity limits the maximum speed-up.

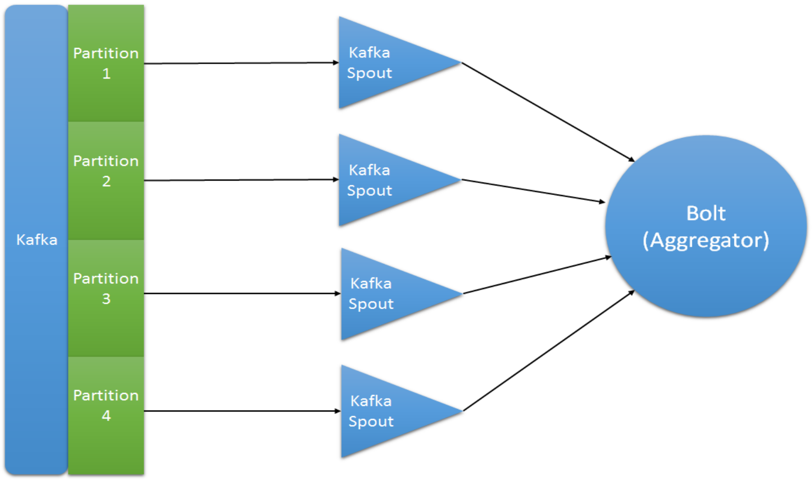

Figure 5: Heron Hashtag Count Topology with 4 node Cluster.

### 3.2.2 Perceptron:

This test case computes the binary classification of tuples based on trained data model. We have used dataset with 20 binary valued features and one bias input (-1). The train set is used to learn the model (weights for each feature). The output is binary valued as well. Test set is used to run Perceptron in Streaming systems. In this test, the tuples will be items with features separated by newline. Similar to Hashtag Count setup, we are feeding these tuples to Apache Kafka and then reading it from Kafka. The number of items in the test set is about 26 million. There are two differences in this test case: First, as this test case doesn't need any cumulative running state in both the systems, hence speed-up with parallelism can be achieved (the cumulative running state, the Hashmap, becomes the bottleneck in previous test case). Second, this test case is more compute intensive than Hashtag Count test. The setup is exactly same as the previous test case with one change in heron that, it has four bolts in the cluster environment as depicted in the Figure 6. Apart from reading in parallel, Spark uses the same previous environment.

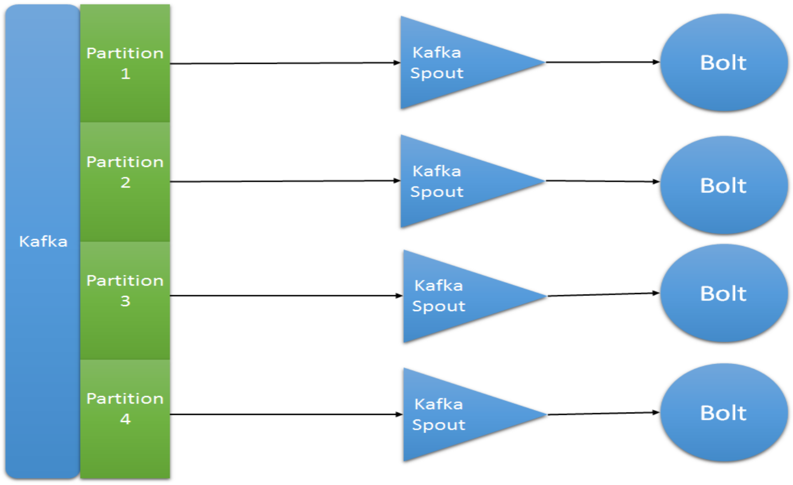

Figure 6: Heron Perceptron Topology with 4 node Cluster.

## 4. Evaluation.

In this section, we discuss the most important part of the project: Performance comparison. For our benchmark, we have chosen **Throughput, Latency,** and **Speed-up** as the main characteristics against which we will compare Spark and Heron. Here, *throughput* is the number of tuples processed per second. *Latency* is the amount of time a tuple took to get processed and produce output; i.e., the time difference between the entry of the tuple into the system and the time of its exit (output). This is basically the turnaround-time or **end-to-end latency**. We should note that this is different from individual tuple latencies/wait times; We have used the ***Average Latency*** (Sum of latency of all tuples divided by the number of tuples) to depict the graphs. From here on, whenever we mention Latency, it is Average End-to-end Latency. *Speed-up* is the measure of how fast the system processes tuples with addition of nodes into the cluster. Both Heron and Spark have various configurations that are depicted in the graphs. Heron has **two** configurations: With ACK enabled (at-lest-once semantics) and ACK disabled (at-most-once semantics). By default, spark provides at-least-once semantics and has **6** configurations: Micro batching interval from 50 milliseconds, 100 milliseconds to 2000 milliseconds. The measurements that we took with only single partition in Kafka, we call **Without Parallelism/No Parallelism.** And we call the measurements in Four partitioned Kafka as **With Parallelism.** For Speed-up analysis, we have collected throughput numbers with Without Parallelism and with Parallelism environments. For each of the test cases discussed in section 3 (Hashtag Count and Perceptron), we have plotted 5 graphs. Each of which represent,

1. **Two Latency vs Throughput graphs** (A Summary graphs): This graph summarizes the performance of Heron and Spark in terms of throughput achieved per second against the latency. There are two graphs per test case, one with Parallelism and one without Parallelism.
2. **Two Throughput Graphs:** This graphs depicts the through put performance of both the systems in their various configurations. Again, two graphs per test case, with and without Parallelism.
3. **One Latency graph (**Latency Summary): This graph shows the latency in micro seconds of both With and Without-Parallelism schemes. Note that this latency is average end-to-end latency.

**Spark Configurations:**

Spark uses DStreams [7] as Data Structure to store inputs from the stream. DStreams are extension of RDD [1] which is basically an immutable set of elements. Spark waits for inputs from stream to fill these DStreams for predefined *batching interval*. Thus, Spark is not truly stream processing system like Heron but emulates via micro-batching of streamed inputs. The length of batching interval affects the throughput. Hence, we have chosen 5 batching intervals of **50ms**, **100ms**, **200ms**, **500ms**, **1000ms** and **2000ms.** One must note here that, with the larger batching interval, the tuples are made to wait for longer period before being processed. Though the throughput is higher with larger batching intervals, we must be cognizant of the wait time of individual tuples (Some application might not accept wait time for some tuples to be 2 seconds for 2000ms batching interval). Thus, we have limited the batching interval at highest of 2 seconds.

## 4.1 Hashtag Count

### 4.1.1 Without Parallelism

The summary graph (Figure 7) clearly shows that Heron with No-ACK configuration outperforms all the other configurations including Heron with ACK enabled. It is tacit that lower latencies increase throughput, and thus it is evident in the graph. No-ACK configuration of Heron has **1.4** microseconds as the average latency and provides throughput of nearly 700,000 tuples per second which is **~4X** of Spark-50 (Spark's lowest) configuration and **2.1X** of Spark-2000 (Spark's Highest). Heron's lowest (ACK) is **1.8X** of Spark-50 configuration and **1.15X** of Spark-2000.

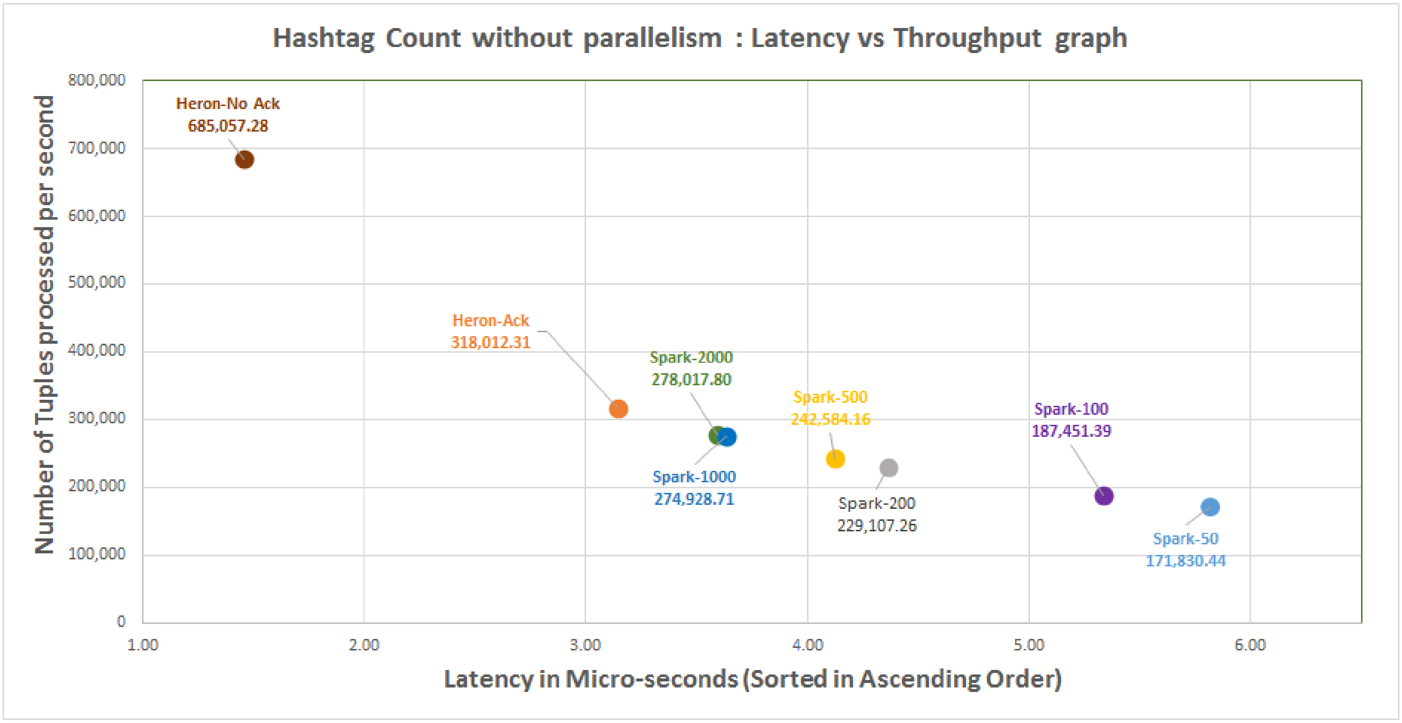

Figure 7: Hashtag Count Summary Graph without parallelism.

The figure-8 depicts the throughput performance; We have measured the time taken by each of the configuration to process 24.7 million tuples. Heron with ACK enabled processes all the tuples in 77.8 seconds whereas No-ACK configuration takes just 36.1 seconds. One important aspect to note here is that, with higher batching interval, Spark performs better. When compared to Spark-50 and Spark-2000, the later perform **1.6X** better than the former. But this comes at the cost of increase in average wait time (spark batches tuples for batching interval time before processing).

### 4.1.1 With Parallelism

In this configuration, we have created four partitions in Kafka so that both Heron and Spark can exploit true parallelism by reading things from Kafka parallelly. The summary graph (Figure 9) shows that Heron with No-ACK configuration, again, outperforms all the others. This result is interesting than previous one because four configurations of Spark (200, 1000, 500, 200) performs better than ACK configuration of Heron. Comparing heron's best with spark's, we see that heron perform **1.3 X** better than Spark-2000 and **1.7X** better than Spark-50. It is important to note that, we cannot really measure the Seed-up in this configuration as all tuples ultimately converge to one processing unit which will calculate the counts. In Heron, we can see that, though we are parallelly reading tuples from Kafka, there is only one aggregator bolt which is bottle neck. Thus, we will explore the speed-up characteristics in the next test case. The figure 10 illustrates the same results. Best of Heron (No-Ack) configuration completes processing of 24.7 million tuples in **20.4** seconds and best of Spark complete the same task in **27** seconds.

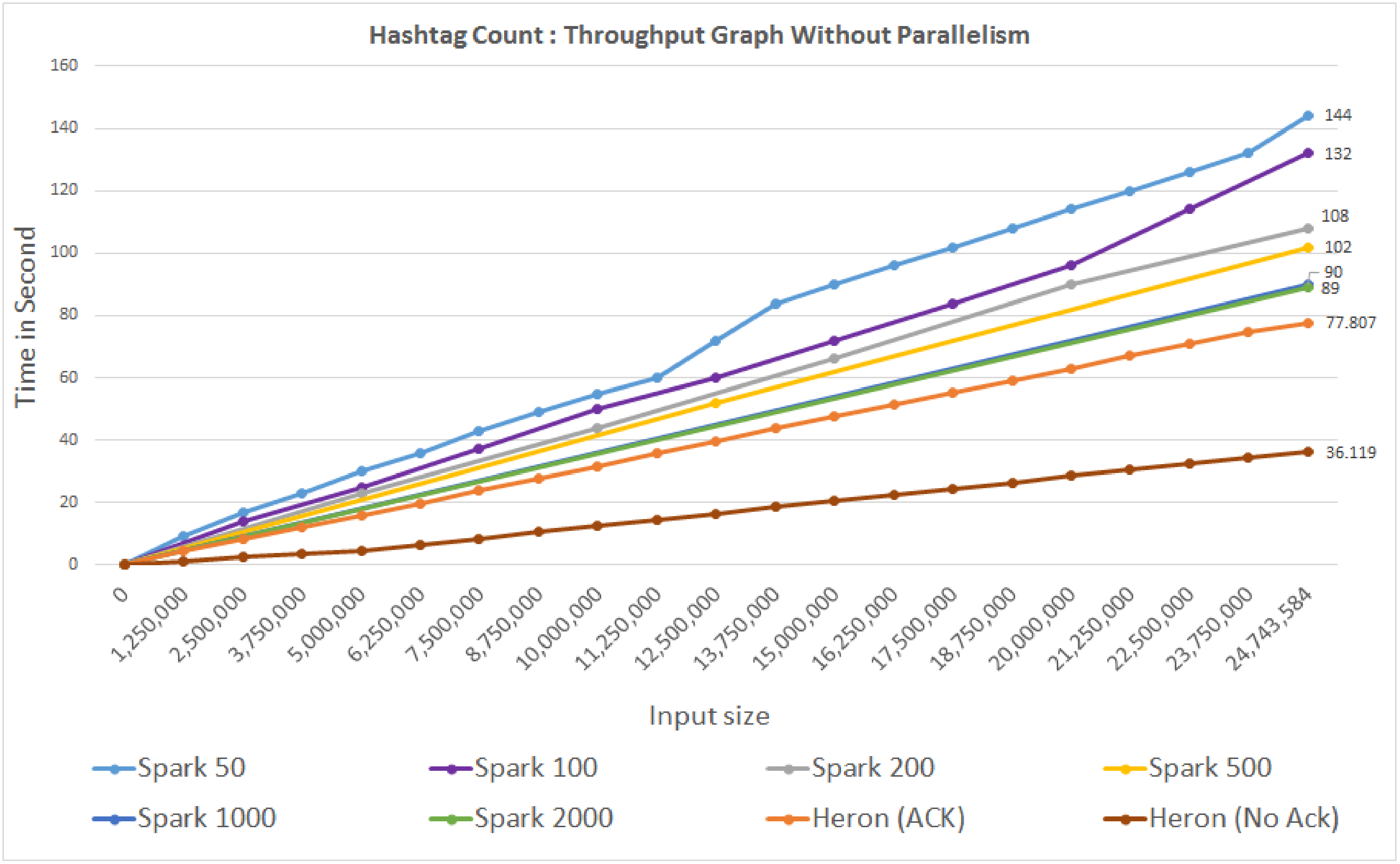

Figure 8: Hashtag Count Throughput Graph without parallelism

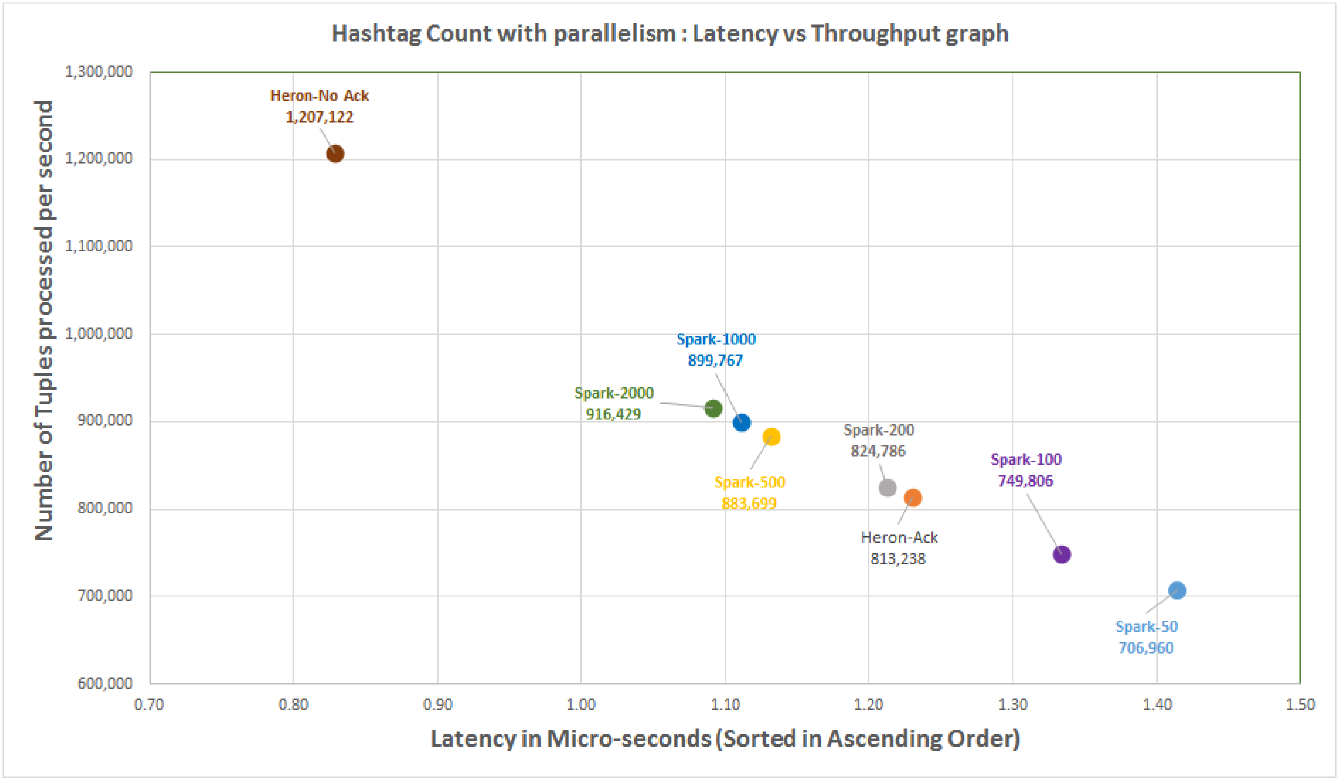

Figure 9: Hashtag Count Summary Graph without parallelism

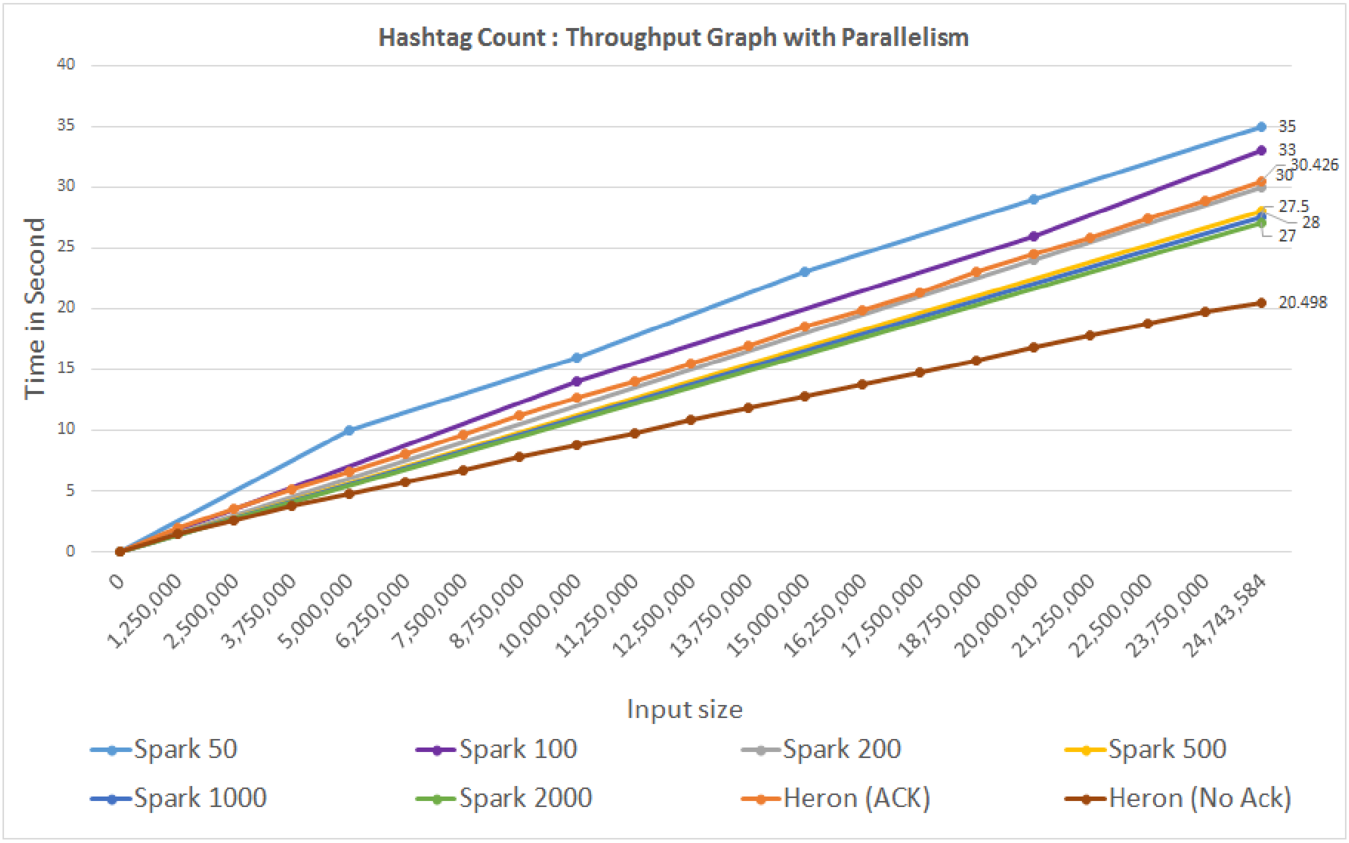

Figure 10: Hashtag Count Throughput Graph with parallelism

Figure 11 depicts the average latencies of configuration in micro seconds. The two bars represent With and Without parallelism latencies; *NO-P* meaning *without parallelism* and *P* meaning *with parallelism.* Like before, no other configurations come close to Heron with No-Ack. The latencies of Heron-ACK with Spark configurations are very close with maximum 200 nanoseconds difference.

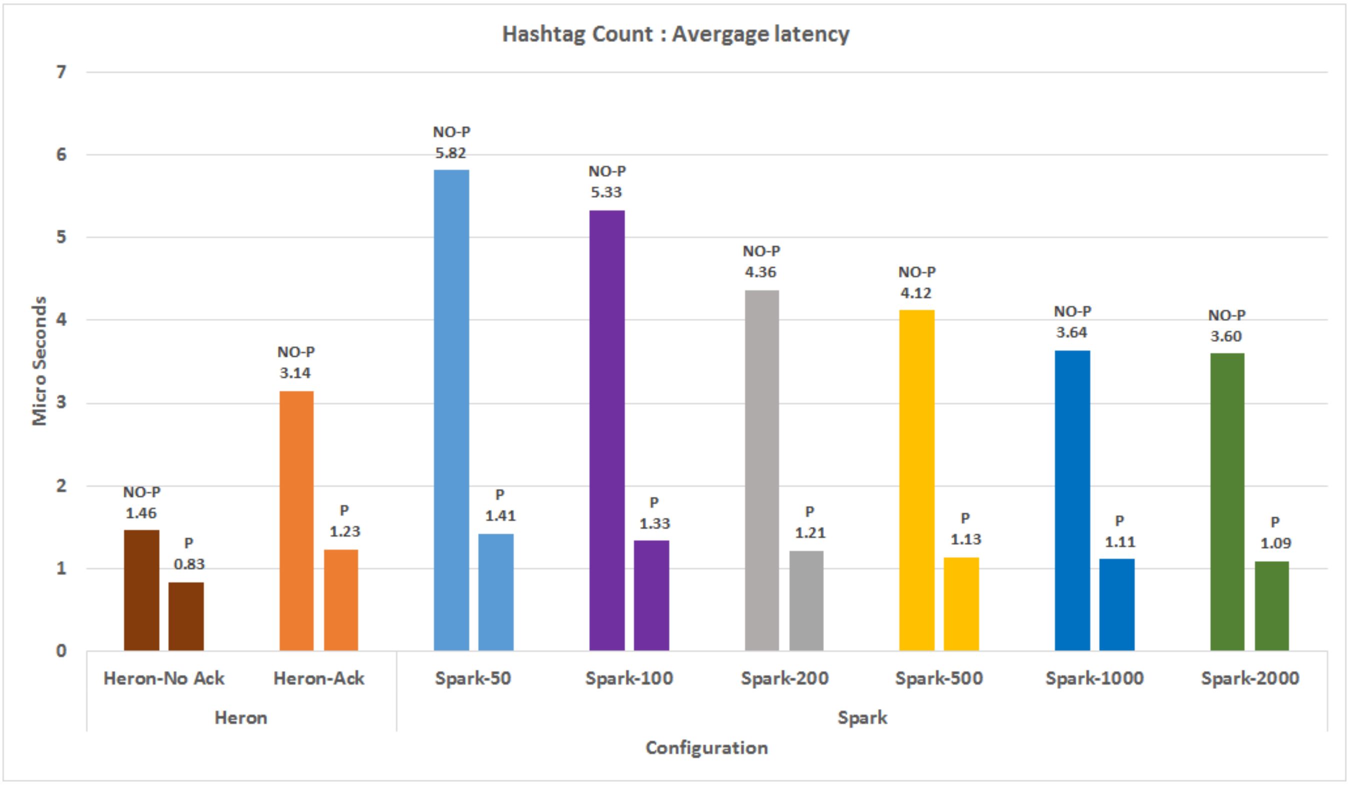

Figure 11: Hashtag Count Average Latency Graph with parallelism

## 4.2 Perceptron

### 4.2.1 Without Parallelism

When compared to Hashtag Count test case, this test case is more computational intensive and it is visible in the graphs. In figure 7, we can see that Heron No-Ack configuration could process 685,057 tuples per second whereas in this test case it is only 89,310 tuples; i.e., 7.6 times less throughput. This shows that the test case is both Receiver and Processor intensive. In Perceptron, Both the configurations of Heron perform better than all the configurations of Spark. Heron achieves **1.2X** to **1.3X** throughput than Spark.

Figure 13 shows the throughput graph of Perceptron without Parallelism. Both Spark and Heron are tested against 26.7 million tuples. Heron-Ack (Least of Heron) configuration takes 314 seconds to process all the tuples whereas Spark takes 370 seconds (Spark's Best, Spark-1000) and 402 (Spark's least, Spark-50). The different is quite big; 56 seconds to 88 seconds. This gap will have a significant impact when processing billions of tuples. One noteworthy points is that, though execute() and nextTuple() methods are called for every tuple in Heron, it is performing lot better than Spark—in which transformations are applied at once after batching.

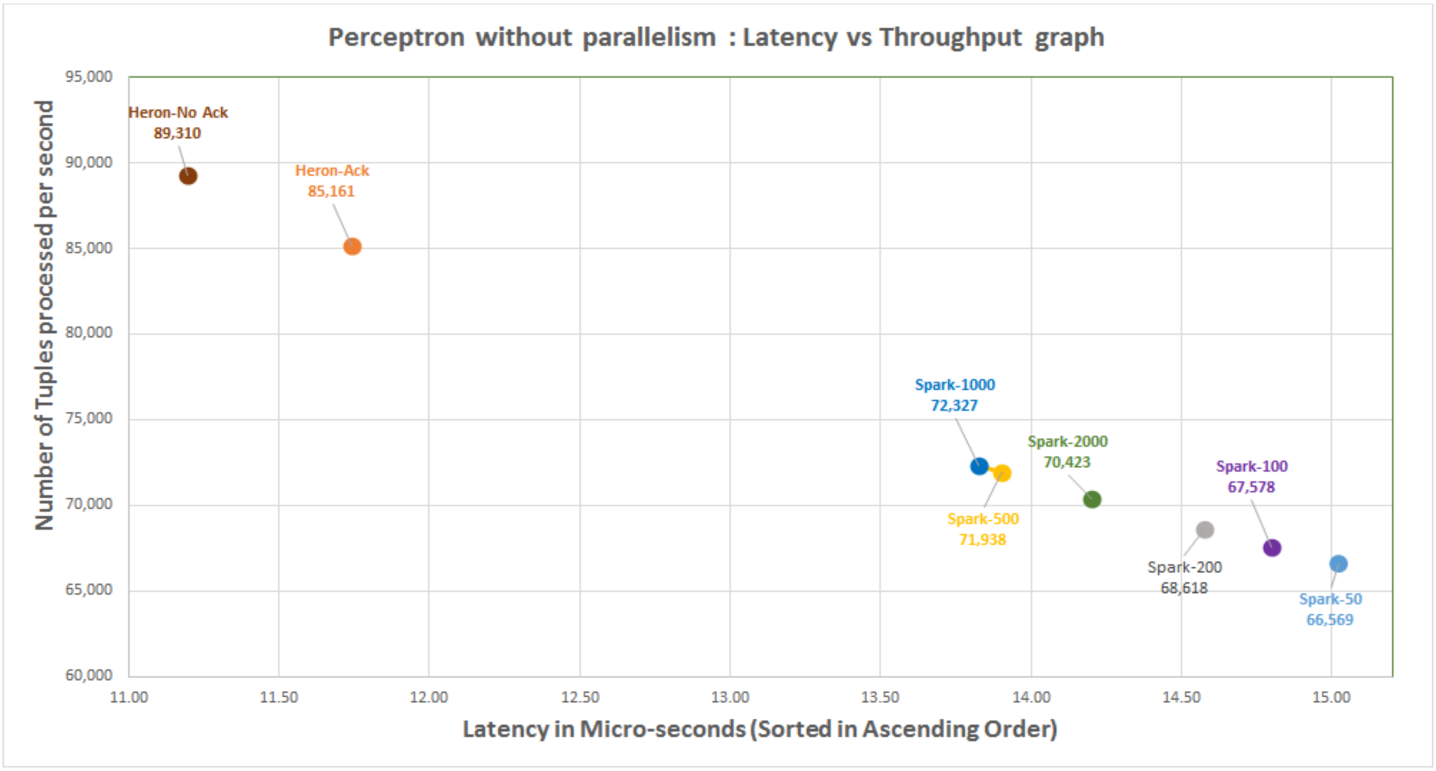

Figure 12: Perceptron Summary Graph without parallelism

### 4.2.1 With Parallelism

Like all previous graphs, comparison details are very similar in this section as well. Thus, we will focus on discussing the other important performance characteristic: **speed-up**. Speed-up can be clearly measure by comparing figure 14 (15) and figure 12 (13). In this clustered environment of 4 workers and 4 Kafka partitions, Spark achieved the ***near-linear*** speed-up of **3.425X** (**85.5%**), whereas Heron achieved ***almost-linear*** speed-up of **3.89X** (**97.25%**)—which is very close to perfect linear speed-up. Heron achieves **1.4X** to **1.58X** better than Spark with parallelism enabled.

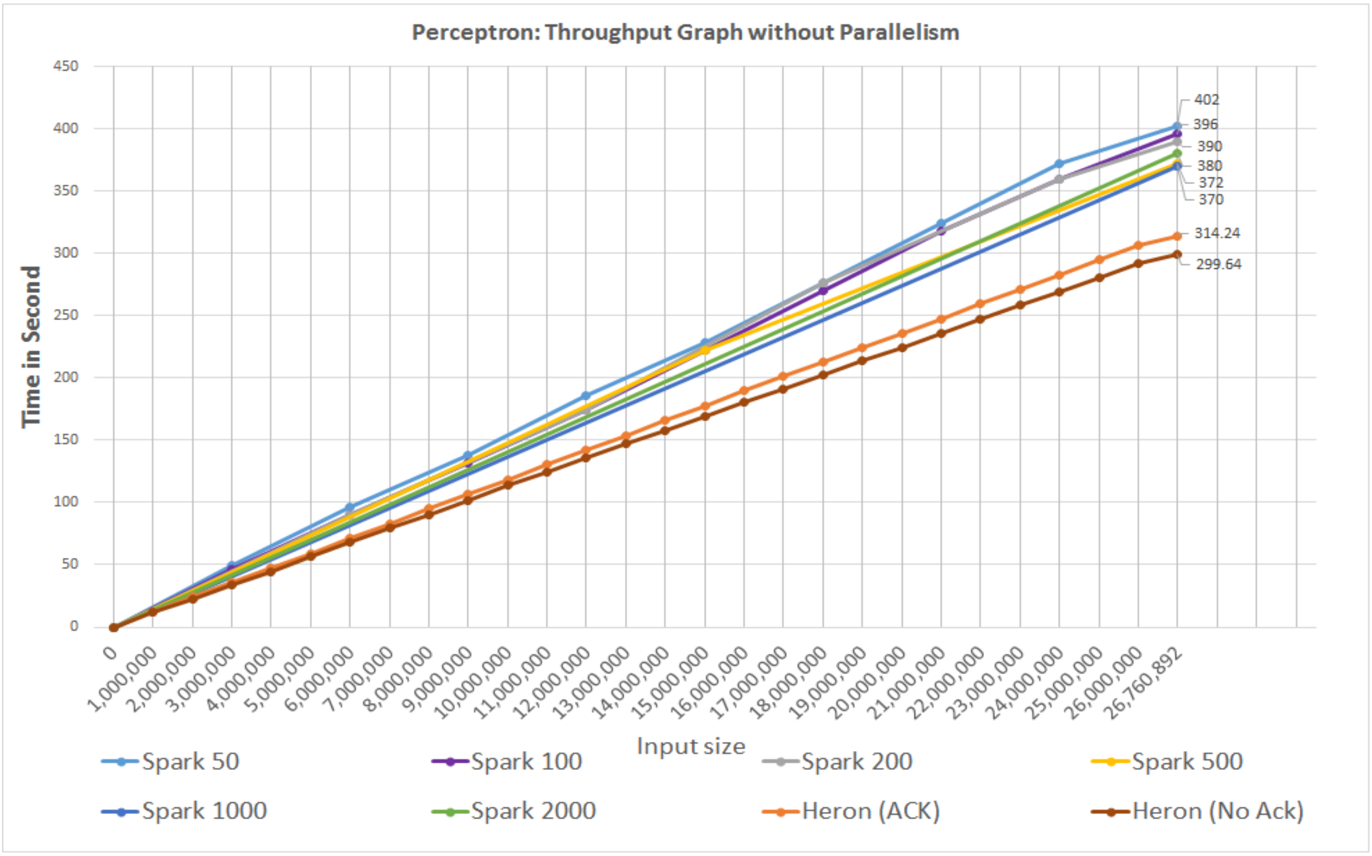

Figure 13: Perceptron Throughput Graph without parallelism

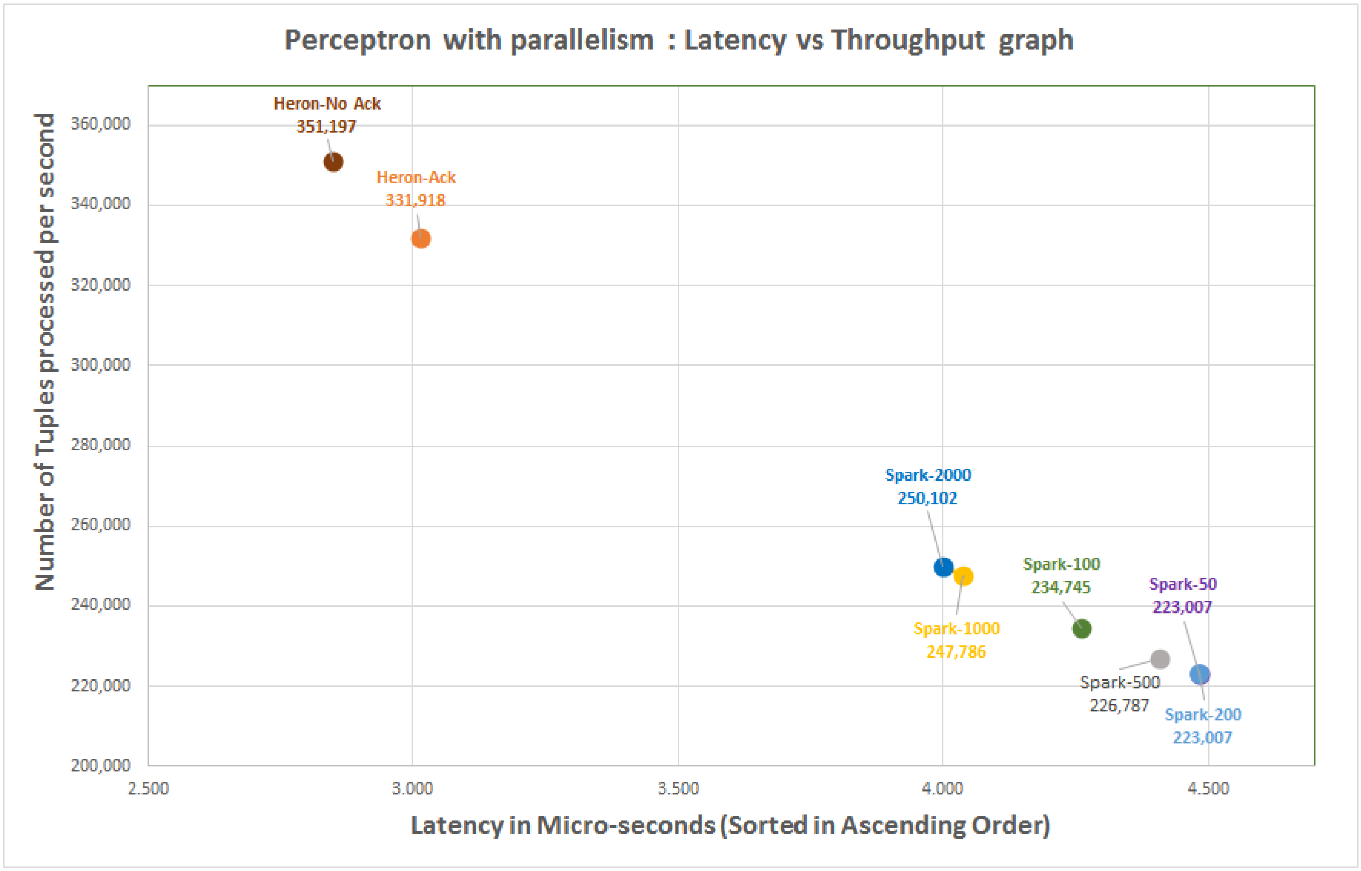

Figure 14: Perceptron Summary Graph parallelism

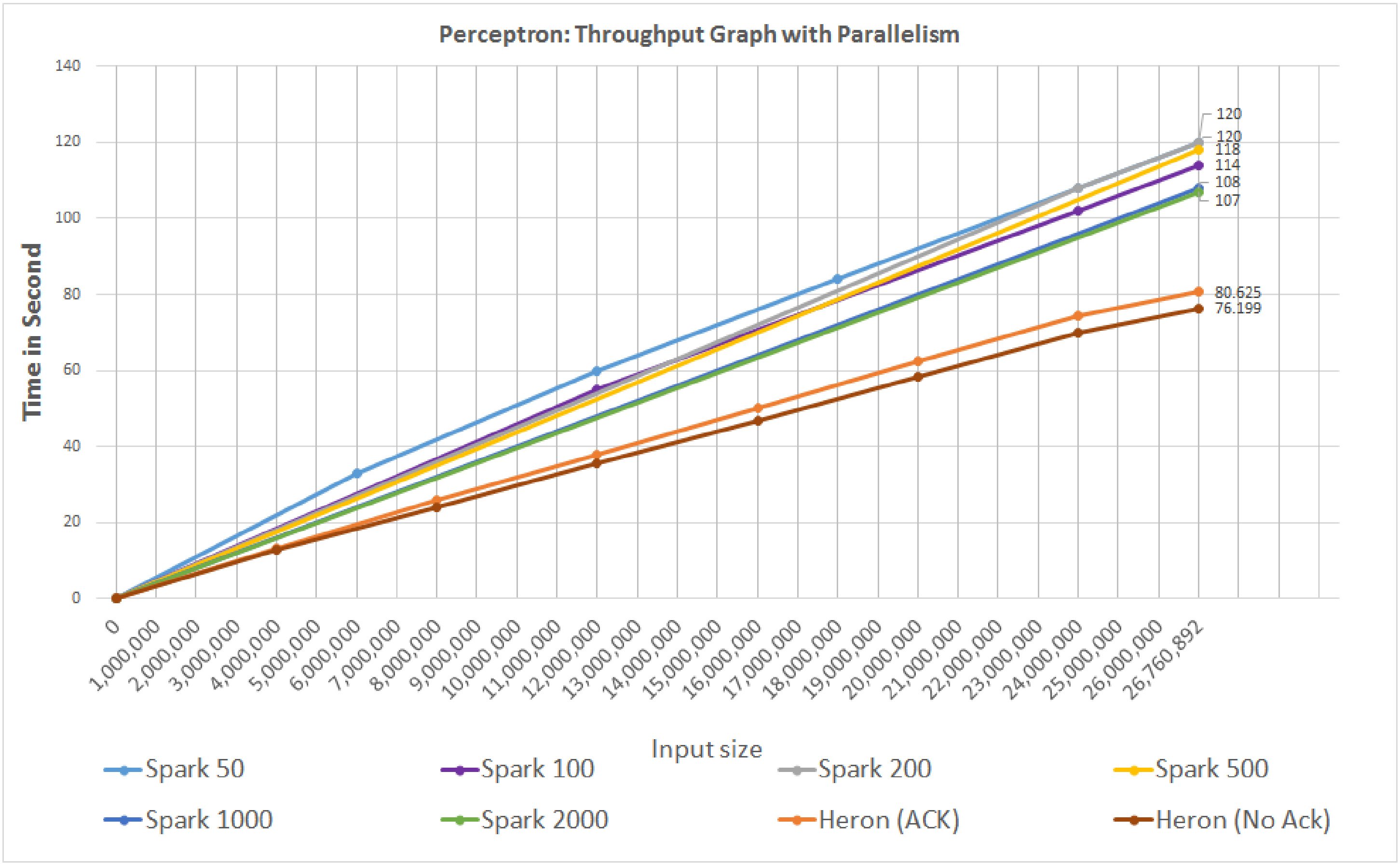

Figure 15: Perceptron Throughput Graph parallelism

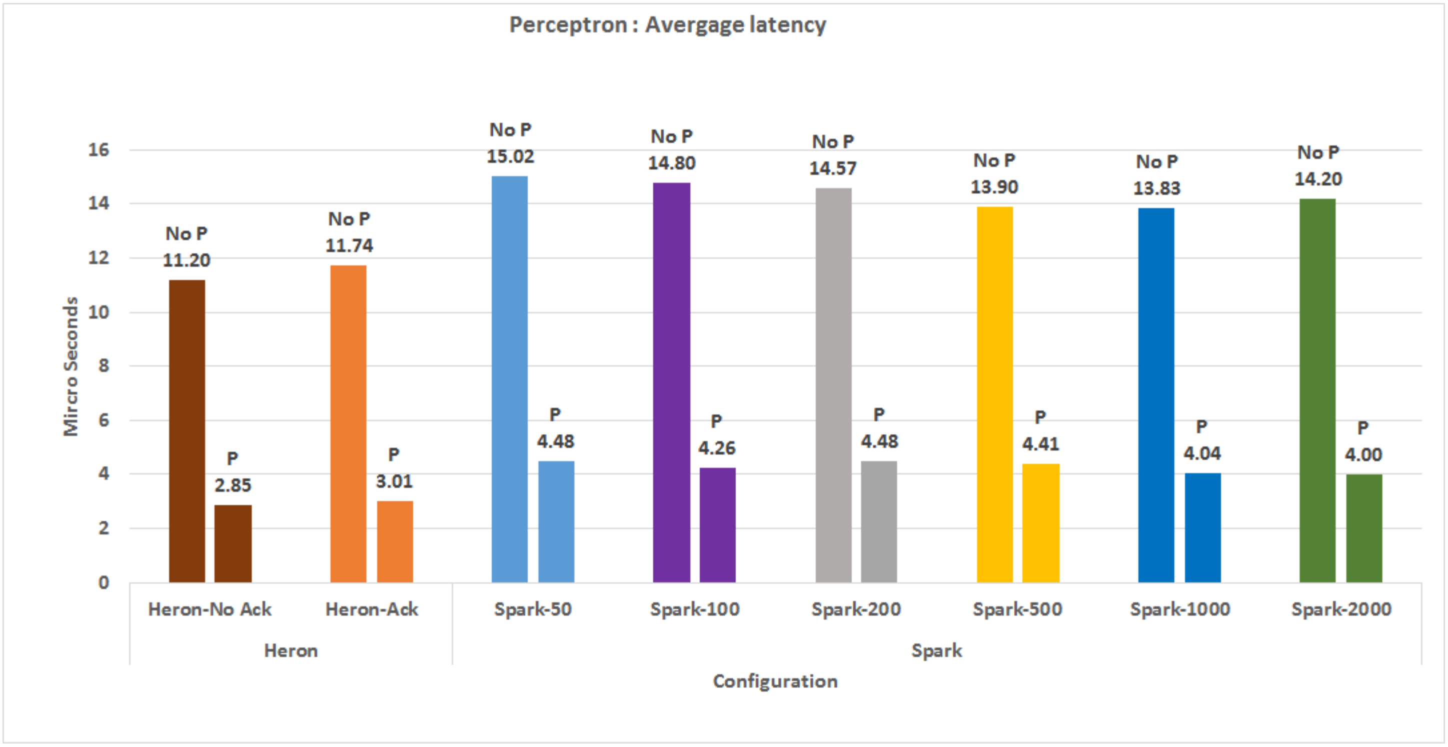

Figure 16: Perceptron Average Latency Graph with parallelism

## 5. Conclusion and Future work:

To conclude, we have successfully evaluated the perform of two Stream Processing System of Spark and Heron. From the graphs, we can understand that Heron was clearly performing better than spark in most of the cases. Below are some of the important takeaways in this report.

- Heron has similar latency for individual tuples whereas Spark's latency depends on its batching interval—that is, with Spark-2000 configuration, first set of tuples, which were inserted in the system, have the wait time of 2000ms. Many of the application might not tolerate this delay in response.
- In spark, to achieve improved (lower) wait-time for an individual tuple, we will need to compromise on throughput and vice-versa.
- The Receiving and processing phases are **not pipelined** in spark, i.e., receiver (spout) and processing (bolt) in spark are run sequentially. In Heron, these two phases are **pipelined**.
- In this clustered environment of 4 workers and 4 Kafka partitions, Spark achieved the ***near-linear*** speed-up of **3.425X** (**85.5%**), whereas Heron achieved ***almost-linear*** speed-up of **3.89X** (**97.25%**)

**Future Work:**

We have compared two types of workloads discussed in section 2 and we need to generate workloads where receiver workload is almost negligible (though not practical, will be able to compare compute part of heron and spark better). We also need to generate workloads to compare the recovery performance, in case of failures, for both the systems. Since failures in large clusters are very common, recovery performance comparison is important as well.

## References

[1] Spark RDD: http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf

[2] Twitter Heron: Streaming at Scale, Proceedings of ACM SIGMOD Conference, Melbourne, Australia, June 2015

[3] Twitter Dataset: https://archive.org/details/archiveteam-twitter-stream-2016-06

[4] Apache Kafka: https://kafka.apache.org/

[5] Spark https://spark.apache.org/releases/spark-release-2-0-1.html
Heron https://github.com/twitter/heron/releases/tag/0.14.5

[6] Streaming-Benchmark: https://github.com/rdamkondwar/Streaming-Benchmark

[7] DStreams: https://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

[8] Yoav Freund.; Robert E Schapire, R. E. "Large margin classification using the perceptron algorithm"

[9] Perceptron: Wikipedia https://en.wikipedia.org/wiki/Perceptron.