Analysis of Real Time Stream Processing Systems Considering Latency

Patricio Córdova
University of Toronto
patricio@cs.toronto.edu

***Abstract*—As the amount of data received by online companies grows, the need to analyze and understand this data as it arrives becomes imperative. There are several systems available that provide real time stream processing capabilities. This document analyzes two of the most notable systems available these days: Spark Streaming [1] and Storm Trident [2]. The two main contributions of this document are: the description and comparison of the main characteristics of both systems and the analysis of their latency. There is a benchmarking showing the time taken by both systems to perform same tasks with different data sizes. The benchmarking is followed by a breakdown of the times obtained and the rationalization of the reasons behind them.**

***Index Terms*— analysis, analytics, apache, benchmarking, big data, comparison, database, database management system, databases, dbms, distributed databases, latency, processing, real, real time analytics, real time stream processing, spark, storm, stream, stream processing, streaming, trident.**

I. Introduction

As more data is obtained by online companies, due to the expansion of internet and massive use of mobile devices, the need to analyze this data is growing significantly. There are hundreds of cases where analyzing incoming data is needed e.g. network monitoring, intelligence and surveillance, risk management, e-commerce, fraud detection, transaction cost analysis, pricing and analytics, market data management, etc. [3]

There are several different categories of systems that facilitate data analysis. One of this categories are real time stream processing systems. If we consider all the data that companies receive, much of it is received in real time and it is most valuable at the time it arrives [4]. There is no point in detecting a potential buyer after the user leaves your e-commerce site, or detecting fraud after the burglar has absconded [5]. In fact, these streams of data are everywhere: TCP streams, click streams, log streams, event streams, etc. Processing and analyzing data streams in real time is a difficult task if we consider data sizes of hundreds of megabytes per second.

Luckily, there are many real time data stream processing frameworks available, capable of processing big amounts of data considerably fast. Some of the main systems available are Twitter's Storm [6], Yahoo's S4 [7], Apache Spark [8], Google's MillWheel [9], Apache Samza [10], Photon [11], and some others. The purpose of this study is to analyze two of the most notable open source platforms already mentioned: Storm and Spark, considering their Trident and Streaming APIs respectively. Both systems offer real-time processing capabilities over micro-batches of tuples but their functioning mechanisms, failure response, processing speed and many other characteristics are different. The purpose of the study is to provide an overview of many of the high level features of both systems and to understand and evaluate their processing speed.

II. The Systems

A. Spark

"Apache Spark is a fast and general-purpose cluster computing system" [12]. The main abstraction in Spark are RDDs [13]. RDDs are collections of objects spread across a cluster and stored in RAM or on disk [8]. These collections are built through parallel transformations that are tracked to keep lineage, which enables fault recovery.

Spark Streaming

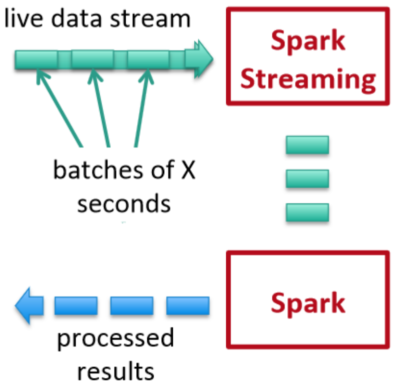

Fig. 1. Spark Streaming operating mode.

The idea behind Spark Streaming is to treat streaming computations as series of deterministic batch computations processed by Spark's core on small time intervals. The mechanism can be seen in *Figure 1*. The data received in each interval is stored across the cluster forming an input dataset for that interval [5]. Once the interval has timed out, the input dataset enters the system to be processed. The results of the computation are stored in RDDs that are grouped together to be manipulated by the user. This abstraction is called discretized streams (D-Streams) [4]. D-Streams provide stateless operations independently on each time interval as well as stateful operators which operate on multiple intervals and may produce intermediate RDDs as state.

B. Storm

“Storm is a real-time fault-tolerant and distributed stream data processing system” [6]. It defines workflows in Directed Acyclic Graphs (DAG’s) called “topologies” [14]. The abstraction for data flow is called stream, which is a continuous and unbounded sequence of tuples [15]. Streams originate from spouts, which flow data from external sources into the Storm topology. The entities that provide transformations over the streams are called bolts. Each bolt implements a single transformation that can be map reduce [16] functionality or more complex actions like filtering, aggregations, communication with external databases, etc.

Storm Trident

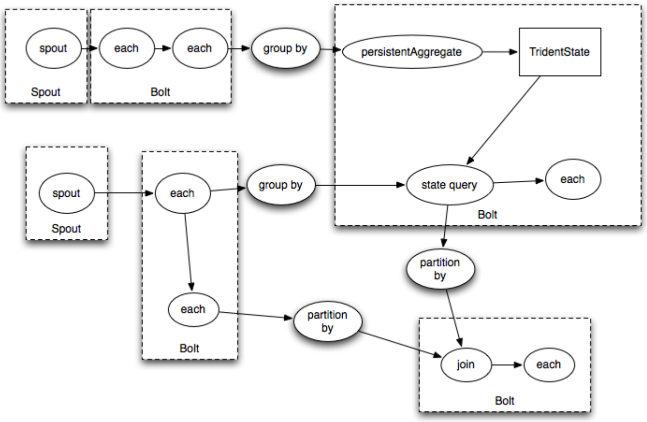

Fig. 2. Trident topology as series of spouts and bolts.

Trident is an abstraction on top of Storm that provides micro-batching and high level constructs like groupBy, aggregate, count, etc. [17] Trident does micro-batching by dividing the incoming streams in small batches of tuples. Moreover, all the high level constructs are basically series of spouts and bolts that are created after compiling the topology. *Figure 2* is an example of a Trident topology seen as series of spouts and bolts.

III. Overview

1) Fault Tolerance

In Spark Streaming if a worker fails the system can recompute the lost state from the input data by following all the RDD transformations that preceded the point of failure [18]. Nevertheless, if the worker where the data receiver is running fails, then a small amount of data may be lost, this being the data received but not replicated to another nodes yet.

In Storm if a supervisor node fails, Nimbus will reassign that node’s tasks to other nodes in the cluster [19]. Due to Storm’s acknowledging mechanism if a tuple introduced to the system has timed out, which means that for some reason (like a worker failed) the tuple couldn’t be processed, it will be replayed. In Trident the whole batch is replayed.

2) Processing Guarantees

Spark Streaming provides exactly-once semantics in the best scenario. This semantics applies only at the time when batches are delivered and requires a HDFS-backed data source [20]. That means that in failure scenarios Spark Streaming can degrade to at-least once semantics. Storm Trident provide exactly-once semantics when the source is durable, this means that a tuple should be replayed in case it is timed out. To update state it uses transactions.

3) Processing Models

Spark is a batch processing system. Spark Streaming provides micro-batching processing adapting the Spark library core for this purpose. Storm is a one-at-a-time processing system: a tuple is processed as it arrives, so it is a true streaming system. However, Storm Trident provides micro-batching, which makes it comparable to Spark Streaming.

4) Programming Models

Both systems have different programming models. In Spark its model relies on data parallelism [21] as all the transformations happen around the data. On the contrary, Storm’s programming model is based on task parallelism [22]. As in Storm all the processing is done through bolts and spouts that work over tasks distributed across the cluster, all the transformations and operation are defined based on tasks.

The main primitives in Spark are RDDs and in Spark Streaming are DStreams (that consist in series of RDDs). Spark’s sources can be HDFS, TCP sockets or custom receivers like Apache Kafka, Apache Flume, ZeroMQ, Amazon Kinesis, Twitter Streams, and others. The computation is done through transformations (e.g. map, filter, groupBy) and actions (e.g. count, collect, save) over the data [20]. Spark provides stateful operators too that are applied over each RDD to persist the state.

In Storm the main primitive is Tuple and in Trident apart of Tuple there are Tuple Batch and Partition. The sources of data are called spouts and the computation is done by bolts. In Trident, as was mentioned before, this bolts are encapsulated into higher level operations like filters, aggregations, joins, functions, etc. In Storm there are not stateful operations so the state must be managed manually in the bolts. However, in Trident the state can be managed by the classes State and MapState.

5) Programming APIs

Both systems allow development in more languages than their core ones. Spark supports development over Java and Scala. However, as Spark demands to use

Scala Tuples when programming in Java, its usage tends to be awkward for people that lack knowledge of Scala/Java compatible structures. Storm supports development over Phyton, Ruby, Scala, Java, Clojure and other programming languages thanks to the Storm's Multi-Lang Protocol [23]. However, Storm Trident allows development over Java, Clojure and Scala only.

6) Implementation Language

Both Spark and Storm are defined in JVM based languages. Spark is written mainly in Scala (81.1%) and is compatible with version 2.10.4 [24]. Scala is a multi-paradigm language with characteristics of functional and object-oriented languages. In fact, many of the distributed operations of Spark "mimic" Scala's operations over lists, maps, etc.

Storm is written in Java (66.6%) and Clojure (19.9%). It is compatible with Java 1.6 and Clojure 1.5.1 [25]. Clojure is a dynamic programming language based on Lisp. It is predominantly functional and provides a robust infrastructure for multithreaded programming which makes it very suitable for Storm's programming model.

7) Installation/Administration

Installation of both systems require basic knowledge of Linux terminal and networks. In the case of Spark its installation is pretty straightforward because it doesn't need to be installed in each worker machine, or YARN or Mesos clusters if that is the case. The only requirement is to run the master daemon and enable SSH passwordless communication between the master and worker nodes [26]. Storm requires to have the Nimbus running in the master machine and to be installed in each worker node running as supervisor. It requires to have Zookeeper installed and running too because it will be in charge of coordinating the topologies. As Storm is a "fail-fast" system it also requires to run under supervision so it must be configured under a supervision service like supervisord [27].

IV. Benchmarking

The next goal of this work is to understand latency in both Spark Streaming and Storm Trident. For this sake, we conducted two experiments in order to have a broader picture of how the systems behave. The first experiment explained in this section is a benchmarking of the systems using the University of Toronto's CSLab cluster [28] and performing three different tasks over two datasets of different sizes. The second experiment was a micro-level analysis to determine where time is spent in the systems and therefore explaining the behavior presented in this section.

A. Experiment

The benchmarking applied to both systems consisted in three different tasks: Word Count, Grep and Top K Words. There were two datasets used of one million rows each: the first was a database of tweets captured before the study from the Twitter's API [29] and the second was a Lorem Ipsum [30] paragraph. In the case of the tweets each record's size was 100 bytes and in the Lorem Ipsum paragraph each row's size was 1000 bytes. The experiment consisted in timing each system in each of the three tasks three times, for both datasets. The time was measured from the moment where each the system started to consume (and process) from Kafka, this being around two seconds from the moment that Kafka began to ingest the datasets. In total there were 18 experiments per system so 36 in total.

B. Hardware

We conducted our experiments on a three node cluster. There was one machine acting as Nimbus/master and Zookeeper host (in Storm's case). There was another machine in charge of hosting Apache Kafka and one final machine acting as worker. We did this in order to isolate all the components of the architecture so that the time obtained by the workers was pure processing time. The detail of the specs of the servers are shown in *Table 1*.

Specs/Machine	**Nimbus/ master**	**Kafka**	**Worker**
CPUs /Cores	2/8	2/16	2/12
CPU Type	Intel X5355	AMD Opteron 6128	Intel E5-2620
Cache	4MB	12MB	15MB
RAM	28G	64G	256G
SPEC CPU Int Rate 2006 (base)	93	222	375
SPEC CPU FP Rate 2006 (base)	59	210	326

Table 1. Specifications of the cluster nodes used for first benchmarking.

As can be seen, the computers were chosen so that the lightest work (coordination) was done by the computer with the lower number of cores and RAM. Kafka host was given the highest number of cores so that it could feed Storm and Spark the fastest possible, and the worker machine was a computer with higher capabilities than the Nimbus/master but closer in characteristics to Kafka's host.

C. Settings

For the experiment, there we chose the latest versions available of Spark Streaming and Storm Trident: 1.1.1 and 0.9.3 respectively.

Dataset	**Tweets from Firehose's API**	**Lorem ipsum paragraph**
# records	10000000	10000000
~bytes	100	1000
Partitions	1	1
Replication Factor	1	1
# Producers	**10**	**25**
Best producing time	**00:00:32**	**00:00:40**

Table 2. Kafka 0.8.1 tuning parameters.

Kafka, Spark and Storm's parameters were tuned so that they could achieve the fastest speed possible in the architecture described. There are several parameters that can be tuned [31] [32], a detail of the configurations that gave the best performance are shown in *Table 2* and *Table 3*.

System	**Spark Streaming 1.1.1**	**Storm Trident 0.9.3**
Job	WordCount	WordCount
Dataset	Tweets from Firehose's API	Tweets from Firehose's API
# records	10000000	10000000
~bytes	100	100
# Workers	**3**	**12**
Parallelism Hint/Level	**Default**	**12**
Best processing time	**00:01:23**	**00:00:56**

Table 3. Spark Streaming 1.1.1 and Storm Trident 0.9.3 tuning parameters.

For Apache Kafka, the summary of the configuration that gave the best performance was to use 10 producers for the 100 bytes dataset and 25 for the 1000 bytes one. Notice that the replication factor was kept in 1 to speed up the producing time because the data loading was performed at the same time Spark and Storm consumed from the data (to simulate a real streaming scenario). If the number of partitions and the replication factor increased, the time that Kafka took to load the data would have being higher than the time taken by Spark and Storm to process it, so the benchmarking would have been done over Kafka instead over the two systems. Also, the number of partitions (that determine the consuming parallelism) was kept in 1. As Storm and Spark have different manners of consuming data from Kafka, the way to make the process fairer and uniform was to assign only one partition. In a production environment the replication factor and the number of partitions are parameters that must be calculated according the failure response and latency needs of the job, but considering the overhead produced by increasing both parameters [33].

For Spark and Storm the parameter that produced the main difference in the response time was the number of workers. In Spark the time variation was quite high, in fact with a very high or very low number of workers [31] it would have taken up to 50% more time to do the same job. The best configuration was to use three workers and maintain the default task parallelism. In Storm the number of workers concurred with the number of cores of the machine (12). The parallelism hint of the tasks gave the best performance with the same number: 12, however, the difference in the time taken to do the job wasn't too high depending on the number of parallel tasks: only 2 or 3 seconds more, varying from 4 or 20 parallel tasks.

D. Results

The results of the experiment can be seen in Figure 3. Each bar displays an average of the three attempts done in the same experiment.

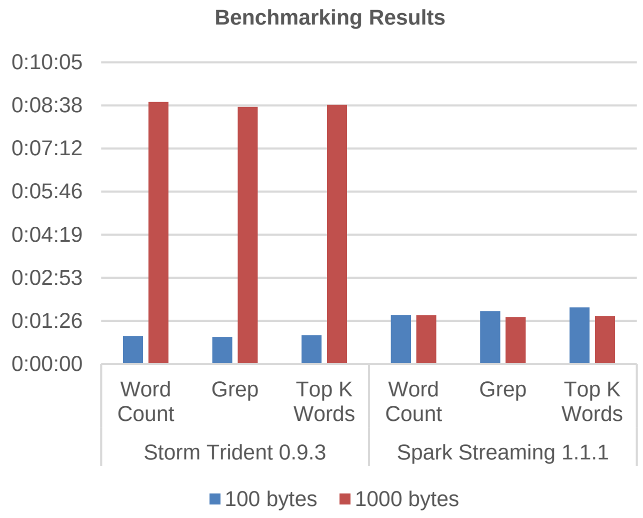

Fig. 3. Time taken by Storm Trident and Spark Streaming to process one million rows in the architecture already proposed in Section V.

As can be seen, first, the times for each system didn't vary much depending on the task (Word Count, Grep, Top K Words). Second, Storm was around 40% faster than Spark processing 100 bytes. However, when the size of each row increased to one kilobyte (ten times) the processing time in Storm grew near linearly. This didn't happen with Spark that keep near the same time for both data sizes.

V. Latency Analysis

The following section describes the second phase of the study that consists in understanding the main components of both systems and how these component's operation affects the overall response time in Spark and Storm.

A. Experiment

The experiment consisted in debugging, profiling and introducing logging calls in the systems in order to track the operation of their main components. For this experiment the job considered was word count.

As was mentioned in the beginning of the document, both systems work over Directed Acyclic Graphs for the topologies and jobs. These DAGs are scaffolds distributed across the cluster and are in charge of all the processing.

In Spark Streaming the DAG generated represent the series of transformations and operations that Spark performs over the data to get the results. In the word count topology the main transformations and operations generated after compiling and debugging the job were the following:

1. SocketInputDStream
2. FlatMappedDStream
3. MappedDStream
4. ShuffledDStream

These classes represent the split, map and reduce operations over the tuples.

Storm Trident generates a DAG of spouts and bolts that represent the Trident topology. As was explained before, Trident supports micro-batch processing and provides high level abstractions. Once the high level topology is defined in the code and it is sent to run, the topology is turned into a DAG of spouts and bolts. In the word count topology the main spouts and bolts generated after compiling and debugging the topology were the following:

1. AggregateProcessor
2. StateQueryProcessor
3. EachProcessor
4. MultiReducerProcessor
5. ProjectedProcessor

These classes are in charge of the split, group by and reduction operations and here is where the logging calls were introduced to see when each job starts and ends.

B. Hardware

As the idea of this experiment was to track every component in the topology/job running on the systems, the analysis had to be done so that the measurement of the time in milliseconds show differences. Spark and Storm ran in local mode in a personal computer with an Intel i7 1st Gen processor with eight cores and eight gigabytes of RAM. Although these characteristics are quite fast for the simplicity of the job, the time was tracked properly and the differences were visible.

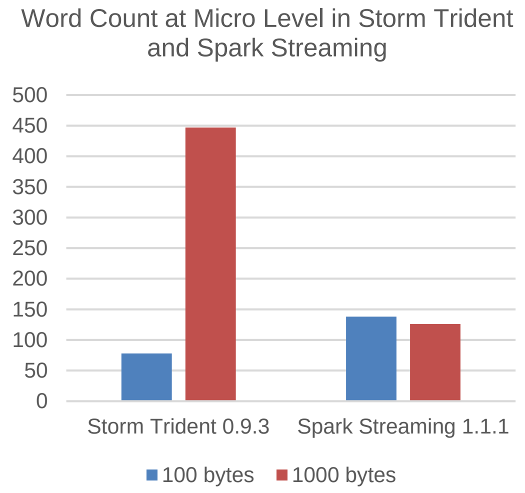

Fig. 4. Milliseconds taken by Storm Trident and Spark Streaming to count the number of words of one tuple of different sizes over the hardware described in Section V.

C. Results

Both systems were fed with one hundred and one thousand bytes of data as in the previous experiment, debugged and all the results kept in logs.

The results obtained are very similar to the ones gotten in the benchmarking of the previous section but at a micro level and the times obtained are split by task. The comparison can be seen in *Figure 4*.

As a summary, the time that Storm Trident took to process 100 bytes was 78 milliseconds and to process 1000 bytes was 447 milliseconds. The most expensive operations (that in the code represent series of spouts and bolts interactions) were splitting and emitting the words and also doing the aggregation: for 100 bytes it took 8 and 32 milliseconds per each, whereas for 1000 bytes took 273 and 132 milliseconds each. There must be considered that in all the operations acking the tuples and keeping state are extra operations that introduce overhead, this behavior will be explained in the next literal. In *Table 4* we show a summary of the times. Notice that many of the tasks happen in parallel so the total processing time is the time reported by the system and not the sum of the partial times.

	Time in milliseconds	
Task	**100 bytes**	**1000 bytes**
Opening and activating spout	5	8
Bolt receives tuple and splits words	**8**	**273**
Saving state result	10	266
Words emitted for DRPC processing, by field grouping	71	441
DRPC aggregation (count)	**32**	**132**
Reducing results	45	424
Total processing time	**78**	**447**

Table 4. Break down of the time taken by Storm Trident to process 100 and 1000 bytes using the hardware described in Section V.

In Spark Streaming the results are very similar to the previous experiment. The time that Spark Streaming took to process 100 bytes was 138 milliseconds and the time it took to process 1000 bytes was 126 milliseconds. The most expensive operations were splitting the words, mapping them to one, and reducing them by key. For 100 bytes Spark took 35 milliseconds to split and map the words, and 80 milliseconds to reduce them. The times were almost the same, even lower, for the 1000 bytes tuples. Notice that in between all these operations, managing the state of the RDD blocks is a recurrent task that incurs overhead. This behaviour will be explained in the next section. Also, as was mentioned with Storm, in Spark many of the tasks happen in parallel, so the total processing time is the time reported by the system and not a sum of the partial times. The details are shown in *Table 5*.

	Time in milliseconds	
Task	**100 bytes**	**1000 bytes**
Generating job after receiving tuple	9	7
Registering tuple in RDD	1	3
Splitting words and mapping to one	**35**	**32**
Reduce by key	**80**	**89**
Total processing time	**138**	**126**

Table 5. Break down of the time taken by Spark Streaming to process 100 and 1000 bytes using the hardware described in Section V.

D. *Explanation*

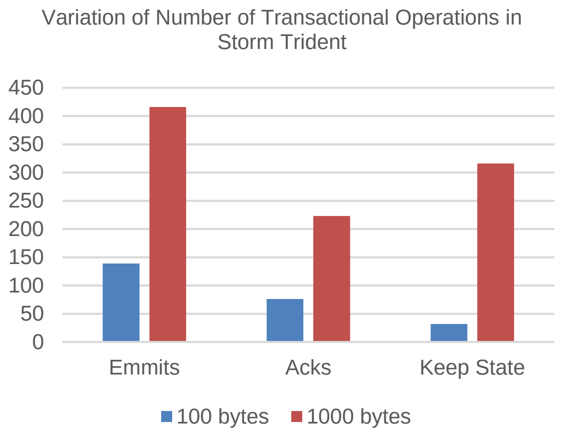

Fig. 5. Number of updates caused by transactional operations in Storm Trident as the tuple's size increases.

It is clear that the times will depend on the cluster, however the important outcome is the understanding of the patterns in the results. First, in the case of Storm Trident, it takes longer to process a tuple as the tuple's size increases. This behavior is attributed to the nature of the system itself: every time a tuple is emitted, it has to go through series of spouts and bolts and it has to be acked and the state must be maintained. This is done through transactions that occupy network resources and of course time. Although all these operations are not as expensive with a small tuple's size (and it is a slightly shorter process than do the processing and maintaining the RDDs in Spark), if the topology gets more complex or the size of the tuple increases, it will take longer to do the processing as it was shown. For example, with the tuple of 100 bytes, the number of acks were 76, whereas with the 1000 bytes the number of acks were 223. The time taken by these operations varies from 0.1 to 1 millisecond. If we consider that for the experiment the number of acks tripled as the tuple's size grew, and the number of operations to maintain the state of the spouts and bolts grew ten times, it is conceivable the linear growth of processing time seen in the experiments. A sample of the variations of the main transactional operations obtained in the experiment are shown in *Figure 5*.

In Spark's case, there are no bolts and spouts, and the tuples flow directly through the DAG without acking. This means, that in the case of word count, the tuple will be split but there will be no further operations, so although the tuple's size increases the time will stand still. However, after any operation is performed, the results are stored as RDDs. The entity in charge of adding, removing and maintaining the RDDs is the BlockManager. This is one of the various components that are in Spark (and not in Storm), that make it take longer than Storm to process a tuple of 100 bytes. For example, the time taken to put and remove RDDs in the experiment for the BlockManager varied from 0.02 to 0.4 milliseconds in average. The growth of the number of times that the BlockManager was summoned as the tuple's size increased rose from 81 times with the 100 bytes tuple, to 124 with the 1000 bytes tuple. As can be seen, the increment is not very significant (around 10ms in total). This is the reason why the processing time was maintained in Spark independently of the tuple's size.

VI. Conclusions

In the current document we described some of the aspects that make Apache Spark (Streaming) and Apache Storm (Trident) two different but very powerful stream processing systems. In Section II and III there we give a brief description of the main characteristics and paradigms of both systems and how they differentiate from each other. Both systems have very similar features so the choice of one over another should be considered carefully and according to the use case, for example, considering processing guarantees, programming models, APIs, etc. In section IV there is a benchmarking of both systems over different tasks and processing tuples of different sizes. The main conclusion of this section was that Storm was around 40% faster than Spark, processing tuples of small size (around the size of a tweet). However, as the tuple's size increased, Spark had better performance maintaining the processing times. In the last section, Section V, there could be seen how all the transactional operations that make part of Storm Trident are the reasons why it increased latency as a tuple's size increased. As future work, a deeper analysis of failure tolerance would give a broader and more complete understanding of both systems. Also, removing some of the transactions introduced in Trident would make it reduce the processing time as the tuple's size increase, so a study to prove this hypothesis would be valuable.

VII. References

[1] Apache Spark. (2014, December) Spark Streaming. [Online]. https://spark.apache.org/streaming/

[2] Apache Storm. (2014, December) Trident API Overview. [Online]. http://storm.apache.org/documentation/Trident-API-Overview.html

[3] Kai Wähner. (2014, December) Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse. [Online]. http://www.infoq.com/articles/stream-processing-hadoop

[4] Matei Zaharia et al., "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," *SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423-438, 2013.

[5] Paul Arindam. SPARK streaming and other real time stream processing framework. [Online]. http://arindampaul.tumblr.com/post/43407070026/spark-streaming-and-other-real-time-stream

[6] Ankit Toshniwal and Et Al., "Storm@twitter," *SIGMOD '14 Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147-156, 2014.

[7] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari, "S4: Distributed Stream Computing

Platform," *ICDMW '10 Proceedings of the 2010 IEEE International Conference on Data Mining Workshops* , pp. 170-177, 2010.

[8] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Spark: cluster computing with working sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* , pp. 10-10, 2010.

[9] Tyler Akidau et al., "MillWheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, pp. 1033-1044, August 2013.

[10] Apache Samza. (December, 2014) Apache Samza. [Online]. http://samza.incubator.apache.org/

[11] Rajagopal Ananthanarayanan et al., "Photon: fault-tolerant and scalable joining of continuous data streams," *SIGMOD '13 Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* , pp. 577-588, December 2014.

[12] Apache Spark. (2014, December) Spark Overview. [Online]. https://spark.apache.org/docs/latest/

[13] Matei Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," *NSDI' 12 Proceedings of the 9th USENIX conference of Networked Systems Design and Implementation*, pp. 2-2, 2012.

[14] ZData Inc. (2014, December) Apache Storm vs. Apache Spark. [Online]. http://www.zdatainc.com/2014/09/apache-storm-apache-spark/

[15] M. Tim Jones, "Process real-time big data with Twitter Storm," *IBM Developer Works*, pp. 1-9, April 2013.

[16] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM - 50th anniversary issue: 1958 - 2008*, pp. 107-113, January 2008.

[17] Apache Storm. (2014, December) Trident Tutorial. [Online]. https://storm.apache.org/documentation/Trident-tutorial.html

[18] Apache Spark. (2014, December) Spark Streaming Programming Guide. [Online]. https://spark.apache.org/docs/1.1.1/streaming-programming-guide.html

[19] Apache Storm. (December, 2014) Fault Tolerance. [Online]. https://storm.apache.org/documentation/Fault-tolerance.html

[20] P. Taylor Goetz. (2014, December) Apache Storm vs. Spark Streaming. [Online]. http://es.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

[21] Wikipedia. (December, 2014) Data parallelism. [Online]. http://www.wikipedia.org/wiki/Data_parallelism

[22] Wikipedia. (December, 2014) Task parallelism. [Online]. http://www.wikipedia.org/wiki/Task_parallelism

[23] Apache Storm. (2014, December) Multi-Lang Protocol. [Online]. https://storm.apache.org/documentation/Multilang-protocol.html

[24] Apache Spark. (2014, December) Apache Spark. [Online]. https://github.com/apache/spark

[25] Apache Spark. (2014, December) Apache Storm. [Online]. https://github.com/apache/storm/

[26] Marko Bonaci. (2014, December) Spark standalone cluster tutorial. [Online]. http://mbonaci.github.io/mbo-spark/

[27] Michael G. Noll. (2014, December) Running a Multi-Node Storm Cluster. [Online]. http://www.michael-noll.com/tutorials/running-multi-node-storm-cluster/

[28] CSLab University of Toronto. (2014, December) Linux Compute Servers. [Online]. http://support.cs.toronto.edu/ApplicationServices/LinuxComputeServers.shtml

[29] Twitter. (2014, December) Public streams. [Online]. https://dev.twitter.com/streaming/public

[30] Lipsum.com. (2014, December) Lorem Ipsum. [Online]. http://www.lipsum.com/

[31] Apache Spark. (2014, December) Tuning Spark. [Online]. http://spark.apache.org/docs/latest/tuning.html

[32] Philip Kromer. (2014, December) Tuning Storm+Trident. [Online]. https://gist.github.com/mrflip/5958028

[33] Michael Noll. (2014, December) Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node. [Online]. http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

[34] Vijay Agneeswaran, *Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives* , 1st ed. USA: Pearson FT Press, 2014.

[35] Pere Ferrera. (2014, December) A practical Storm’s Trident API Overview. [Online]. http://www.datasalt.com/2013/04/an-storms-trident-api-overview/