Conceptual Survey on Data Stream Processing Systems

Guenter Hesse* and Martin Lorenz †
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
e-mail: *guenter.hesse@hpi.uni-potsdam.de, †martin.lorenz@hpi.uni-potsdam.de

***Abstract*—The present paper gives an overview about the state of the art technology within the area of data stream processing systems. Although the area of stream processing systems is not new, it is receiving a greater interest in the light of current business trends like the Internet of Things (IoT). The comparison of systems thereby includes several aspects such as a look into their architectures as well as into the responsibilities of the corresponding system components. A ranking or recommendations for one or more system(s) is not part of the work.**

***Keywords*—Stream Processing; Storm; Flink; Spark; Samza**

I. Introduction

Current trends like IoT comprise various aspects that affect the way of doing business as well as the requirements on software systems. One of the first projects that is related to the IoT idea is the work on a cross-company radio-frequency identification (RFID) infrastructure. It took place at the Massachusetts Institute of Technology (MIT). Additionally, the former executive director of the responsible MIT center, Kevin Ashton, used the term Internet of Things at first in a presentation he conducted in 1999 [1], [2].

No matter which business area we look at in the context of IoT, they all have fundamental ideas in common. Be it the field of predictive maintenance, which describes the continuous condition monitoring in order to predict when maintenance should be done to avoid failures, or smart manufacturing, which describes the improvements of production through IoT, one central part is connected things that collect data.

Using sensors or similar technologies enables to send those data about things or their environment to different destinations. Through collecting and analyzing this data new ways of doing business emerge. Having such comprehensive data allows to act faster and better on certain situations and so to be more flexible, assuming working with real time data.

The mentioned business trends moreover lead to changed and new requirements with regard to information systems. One of those is handling data streams, wherefore data stream management systems can be used. Although systems for managing data streams is not a recently developed concept, it is becoming more important in the mentioned context of IoT. Hence, research and development in that area is going on and of great interest against the backdrop of the challenges related to the above-described business trends.

The objective of this paper is to analyze and compare certain data stream processing systems regarding their architecture and certain aspects that are presented later on. The results shall allow a better assessment and improved comparability of those systems and can be used for further research within that domain. Moreover, it might help choosing a suitable system for IoT application developers that are dealing with data streams.

After giving a short introduction into the area of data stream processing systems, a certain set of actual systems of this type is analyzed in detail. Afterwards, the comparison results, conclusions and an outlook are presented.

II. Data Stream Processing Systems

Stream processing systems or data stream management systems (DSMS), which is used synonymously within this paper, are supposed to handle data streams. Such a stream is a continuous flow of incoming data records, comparable to a data feed. The data might be, for example, sensor data, network data, stock prices or postings in social networks.

Contrary to database management systems (DBMS), data is usually not persisted before query execution, but could be done afterwards. This processing model in DSMSs has an impact on the executed queries. Particularly, so called continuous queries are used. Those are defined and persisted queries that are continually executed using a certain data set [3], [4].

Those data sets, on which a stored query is executed on, are defined by a trigger function. That function specifies when a query processing is supposed to happen. Concretely, a query can be triggered based on factors such as time or the number of newly approached data tuples. Although executing queries on sliding windows of data might give fewer insights compared to an evaluation of a database table with more data, there are advantages of using this technique. Since only the newest data is considered, there is an emphasis on those recently collected values, which makes it easier discovering appearing trends earlier. Additionally, working on a relatively narrow data set reduces the needed amount of memory and hence, it can help preventing memory overflows and saves costs [3], [5].

A challenge for DSMS is the order of the incoming data tuples, which can not be controlled. Of course, there is a natural order, simply by the time of arrival, but that order does not necessarily correspond to the chronological order of the value occurrences. Due to varying factors such as latency or network connection for moving sensors, the data sequence could be upset or values could get lost, which is overall

(Accepted at IEEE 21st International Conference on Parallel and Distributed Systems 2015, http://dx.doi.org/10.1109/ICPADS.2015.106) DOI: https://doi.org/10.1109/ICPADS.2015.106

referred to as a noisy data stream [6], [4]. Consequently, there is a trade-off for certain scenarios between performance and correctness as one can either wait for missing values for an undefined time or execute the query without the missing values or with unintended values.

With respect to the performance-related area of benchmarks, there are no well-established ones for the analyzed group of distributed data stream processing yet, though the work presented in [7] describes itself as "an early step" [7] towards that direction.

III. Comparison of Stream Processing Systems

The existing stream processing systems include older and discontinued projects as well as newer ones. Moreover, there are many open-source projects and several commercial systems. The present work only considers projects belonging to the group of open-sourced projects with an active group of researchers and developers. That is done in order to guarantee as much as possible transparency, comparability as well as relevance to the current state of the art. Particularly, the group of compared systems includes Apache Storm, Apache Spark with its streaming functionality, Apache Samza and Apache Flink. The group of older DSMS with discontinued development contains projects such as Aurora [8], STREAM [9] and Apache S4 [10].

The dimensions that are used for comparison are, next to the general architecture of each system, the used programming languages, message treatment, latency, throughput as well as the kind of message processing assurance.

The programming language can be of interest for IoT application developers as running the system might require additional software installed on the corresponding server(s) such as Java. Furthermore, it becomes relevant if adaptions to a DSMS itself are intended. The kind of message treatment can gather interest since a batch processing model might come at the cost of a decreased latency or restrictions to a limited set of usable operators [11], [12].

Throughput in the context of stream processing systems is defined "as the number of external input messages that are completed per time unit" [13]. In contrast, latency is the time difference between consuming a message and the end of its processing [13]. So all in all low latency and high throughput are desirable. Both attributes are especially relevant for time-critical IoT applications as they determine the performance of stream processing.

The message processing assurance is an important aspect for as it can impact the overall results and so their correctness. So if each message must not be processed more than once, applications may have to think of how to realize that in case the used system does not assure exactly-once-processing [14].

A. Apache Storm

Nathan Marz originally developed Apache Storm at BackType. In 2011, Twitter acquired BackType and in 2012 the project was open-sourced [15]. Storm can be characterized as "real-time fault-tolerant and distributed" according to [15]. It is mainly written in Java as well as Closure [16], treats messages as streams and it uses ZeroMQ, which is also known as ∅MQ, 0MQ or zmq. This component is an open-sourced and comparatively lightweight messaging framework [15], [17].

Persistent queries in the context of Storm are called topologies. A topology is defined as "a directed graph where the vertices represent computation and the edges represent the data flow between computation components" [15]. There are two distinct kinds of vertices. One type is named spout and represents a source of data tuples that is used within the topology. Such a spout could, for example, pull data from a message queue that is forwarding sensor data. The other vertex type is named bold and responsible for processing incoming data. The resulting tuples can be passed to a following set of bolds and so be processed further [15].

A schematic overview of Storm is displayed in figure 1. A Storm system contains three different node types. One of those is the single master node, which is named after the daemon that is running on it - Nimbus. It receives topologies from connecting clients and manages their execution. That includes the distribution and scheduling of execution on the cluster nodes as well as an overall progress monitoring with regard to the throughput. Besides, its architecture follows a fail-fast approach. In case a node gets killed, Storm uses a mechanism that allows its restart without interrupting any topology execution [15].

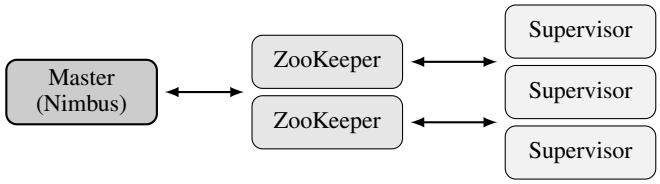

Fig. 1. Architecture of a Storm System (own figure based on [15])

Next to a single master node, Storm contains a cluster of Apache ZooKeeper nodes. ZooKeeper can be seen as "a service or coordinating processes of distributed applications" [18]. In the context of Storm, ZooKeeper knows, next to the local disk(s), about the state of a topology as well as of the master node and the Supervisor nodes, which are described later on. Additionally, as shown in figure 1, ZooKeeper acts as a transmitter of all communication between the other two stateless node types [15].

The mentioned Supervisor or worker node represents the third node type. In order to keep an overview about the state of all Supervisor daemons, a heartbeat mechanism is used. Concretely, each Supervisor periodically sends a heartbeat signal to the master node as well as information about possibly free resources [15]. The main task of a Supervisor is spawning worker processes based on the instructions it gets from Nimbus. That covers checking the condition of the created workers using again a heartbeat mechanism. In case a worker process terminates unexpectedly, the Supervisor restarts the corresponding worker [15], [19]. The overall structure of a worker is conceptually shown in figure 2.

Each worker runs a Java Virtual Machine (JVM) on the its Supervisor and performs a defined part of exactly one topology. That is done by running at least one executor, whereby each executor again consists of one or more task(s) as it can be seen in figure 2. These tasks eventually perform the actions

defined by a spout or bold. In addition to creating stateful worker processes, a Supervisor monitors those and regenerates workers if necessary [15], [19].

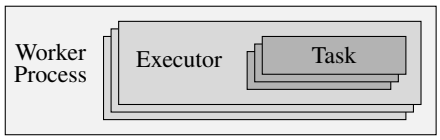

Fig. 2. Architecture of a Worker Process (own figure based on [15], [20])

Storm only assures an at-least-once processing of each message. When restarting a failed spout instance, it continues from its lastly saved state, which is stored in Zookeeper. Through Trident, an abstraction on top of Storm that works on transactions, a stateful stream processing as well as an exactly-once processing can be achieved [11], [15], [21].

With respect to performance aspects, Storm usually has a lower throughput as well as a lower latency compared to Spark Streaming, which is presented later on. Depending on the versions and the used algorithms the results may differ [7].

B. Apache Flink

The root of Apache Flink is Stratosphere, an open-source research project or system for big data analytics. Flink supports batch processing as well as the processing of data streams and can guarantee an exactly-once-processing. It is mainly written in Java as well as Scala and has client APIs for these two programming languages [22], [23], [24], [25].

A Flink Runtime is conceptually presented in figure 3. Similar to the previously presented Storm, the Flink Runtime makes use of the master-worker pattern. Concretely, it consists of two different element types, a Job Manager, the master, and one or more Task Manager(s), the worker(s). So contrary to Storm's architecture displayed in figure 1, there is no layer in between like ZooKeeper [15], [22].

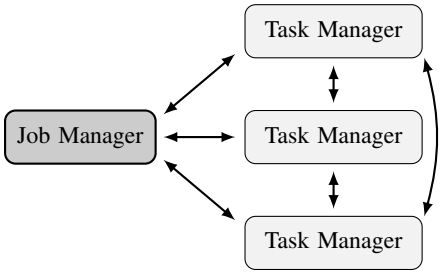

Fig. 3. Architecture of a Flink Runtime (own figure based on [22], [26])

The Job Manager is the interface to client applications and has similar responsibilities as Storm's master node. In particular, those include receiving assignments from clients, scheduling of work for Task Managers as well as keeping track of the overall execution status and the state of every worker. The lastly mentioned task is again, compared to Storm, realized using a heartbeat mechanism. The Task Manager instances execute the assigned tasks or subtasks and exchange information among workers in case that is needed [22].

Each Task Manager provides a certain number of processing slots to the cluster, which are used for parallelizing tasks. The number of slots can be configured, whereby it is recommended to use as many slots as there are CPU cores in every Task Manager node. The degree of parallelism a job eventually uses can be defined in multiple ways [27].

With regard to performance, Flink is considered to have a very low latency that is as low as Storm's according to [25]. Moreover, its throughput is described as high [25], [28].

C. Apache Spark

Similar to Flink, Spark also started as a research project and later on acquired the Apache incubator status. Concretely, Spark originates at the University of California Berkeley (UC Berkeley) and can be seen as a framework for distributed data processing [29], [30]. For covering special scenarios, Spark supports several libraries that are built on top of it. One of those is Spark Streaming [31], a library for data stream processing. ilidistributedmem(The Spark project mainly contains Scala, Java as well as Python code and provides APIs for these languages [32]. A fundamental pattern that is used within Spark is resilient distributed dataset (RDD), which is a memory abstraction. Particularly, an RDD is a read-only "collection of Java or Python objects partitioned across a cluster" [30], [33].

The structure of Spark in cluster mode is displayed in figure 4. The focus within the present paper is on this cluster deployment mode, since it is seen as the most suitable way of using Spark for large-scale data processing. On the left-hand side in figure 4, the Driver Program is shown, which is the application that is executed on top of Spark. Though there might be multiple applications distributed within a cluster, only the main program creates a so called SparkContext. This object is responsible for coordinating the perhaps existing multiple client processes. Furthermore, it is typically and especially in cluster mode connected to a Cluster Manager [20].

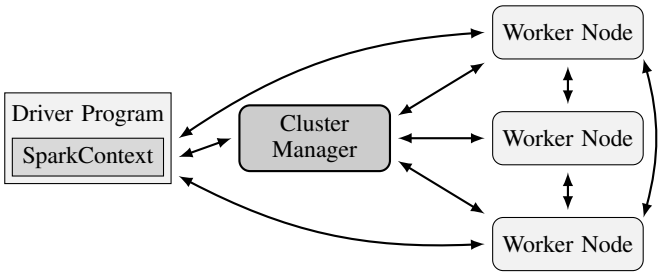

Fig. 4. Architecture of Spark in Cluster Mode (own figure based on [20])

There are three Cluster Managers supported by Spark - the included manager Spark Standalone, Apache Mesos [34] and Apache Hadoop YARN (Yet Another Resource Negotiator) [35]. Its main task is providing executors to applications as soon as a SparkContext has established a connection [20].

One or more executors run in a Spark worker. Equivalent to Storm's worker structure, a Spark worker can run multiple executors and each executor contains one or more tasks. Thus, the conceptual overview of a worker's architecture displayed in figure 2 is also valid for a Spark worker [20].

The Worker Nodes or its executor processes are again responsible for the calculations. Executor processes only work for one program at one time and stay alive until it has finished.

A consequence of the first aspect is a complexity reduction with regard to task scheduling, since each application can schedule the tasks of their exclusive executors independently, meaning without considering other programs. The execution scheduling is done by the Driver Program [20], [33].

This kind of isolation is similar to the described Storm mechanism, where even every worker process is exclusively connected to a single topology, which is equivalent to an application in the context of Spark. A downside of this concept is related to the data exchange between different programs or SparkContext objects, which can only be done through indirections like writing data to a file system or database [20].

The stream processing feature of Spark that comes with its library uses a slightly adapted version of a general Spark system. It conceptually works as shown in figure 5. The data stream on the left-hand side is the input for Spark Streaming, which creates batches out of the stream in form of RDDs. These batches are passed to the Spark Engine, which is doing the calculations [31].

Furthermore, an abstraction for data streams called discretized streams (D-Streams) is used in Spark. Such a D-Stream object consists of an RDD sequence, whereby each RDD contains data of a certain stream interval. The idea for developing this model is providing a better handling for faults and slow nodes within a cluster. In particular, calculations shall be structured "as a set of short, stateless, deterministic tasks instead of continuous, stateful operators" [31]. These calculations in form of small batch computations allow an earlier identification of the mentioned issues and provide an exactly-once message processing. The concept distinguishes Spark from systems that are processing data as a stream and so using the "continuous operator model" [31] such as Storm and Flink. On the one hand, using those micro-batches brings the mentioned advantages in the area of recovery, but on the other hand it increases the latency for message processing [31].

Next to the higher latency compared to Storm and Flink, its throughput can be seen as comparatively high and so higher as, for example, Storm's throughput [7], [28], [31].

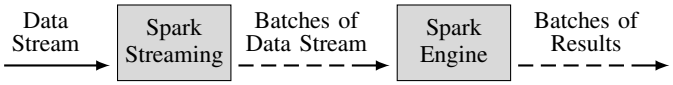

Fig. 5. Stream Processing in Spark Streaming (own figure based on [31])

D. Apache Samza

Apache Samza is a distributed stream processing system mainly written in Scala and Java that treats messages as streams. Overall, it has a relatively high throughput as well as a somewhat increased latency compared to Storm. In Samza's usual setup, two other Apache projects are used, which can be seen in figure 6. The two lower components, YARN and Apache Kafka, represent the two commonly used but exchangeable projects for Samza's execution (YARN) and streaming (Kafka) layer [28], [36], [37], [38].

YARN [35], which is mentioned before as a supported cluster manager for Spark, can be seen as "the next generation of Hadoop's compute platform" [35]. Apache Hadoop is a framework for distributed computing. Particularly, YARN is used for tasks like job scheduling and the management of cluster resources [35], [39].

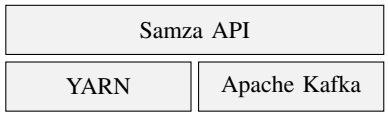

Fig. 6. Common Components of a Samza System (own figure based on [36])

YARN's architecture is illustrated in figure 7. A central component that exists once per cluster is the Resource Manager, which is realized as daemon process running on a dedicated machine. This interface for client applications and also monitors the cluster node's status. Moreover, it decides about resource distribution among applications. It thereby allocates and leases resources in form of so called containers. A container, which usually stands for a UNIX process [36], can be seen as "a logical bundle of resources (e.g., <2GB RAM, 1 CPU>) bound to a particular node" [35].

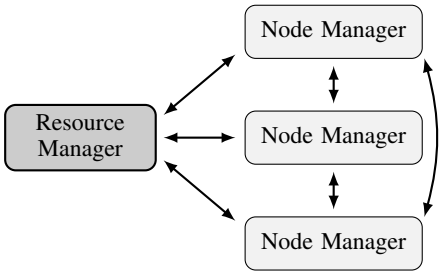

Fig. 7. Architecture of YARN (own figure based on [35])

For managing cluster resources, a Node Manager daemon is running on each node. The tasks of that worker process cover keeping track of the node's resources and notifying about failures should the occasion arise. The communication between the Resource Manager and the Node Manager processes is again implemented using the heartbeat concept [35].

Additionally, there might be communication between the Node Managers themselves as shown in figure 7. However, such message exchange can only happen on application level. Particularly, it can happen between a so called Application Master and its assigned containers. After the Resource Manager accepts an application, it allocates a container and starts the Application Master within it. This master process also sends heartbeats to the Resource Manager [35].

The Application Master manages the program execution regarding aspects such as resource needs and fault handling. That covers coordinating the logical execution plans by requesting resources and generating the physical execution plans accordingly to the actually assigned resources. In order to get new resources, an Application Master has to send a request to the Resource Manager. As soon as a resource lease on behalf of an Application Master is created, the corresponding container is pulled by the Master's next heartbeat. With respect to Samza, adapted implementations are used, specifically a Samza Application Master and Samza Containers [35], [36].

Kafka as the streaming layer within Samza is a distributed publish-subscribe messaging system that was originally developed at LinkedIn. Its initial objective is to collect and deliver

high volumes of event data, in particular log data, with a low latency. Moreover, Kafka and therewith Samza provides an at-least-once message processing guarantee [40].

A Kafka system's architecture is comparatively simple as it only consists of a set of Brokers. Data streams are defined by topics, which are divided into partitions that are distributed over the Broker instances. One can publish messages as well as subscribe to those for retrieving the corresponding messages using a pull mechanism [40].

IV. Comparison Results

An overview of the compared systems regarding the defined and presented aspects is shown in table I. One similarity is the language the systems are mainly written in, which is either Java or Scala. Consequently, every system runs within a JVM and advantages, like the platform independence [41], and disadvantages, such as the performance compared to other languages [42], with respect to the use of a JVM hold true for every system.

TABLE I. Summarized Overview of the Compared Data Stream Processing Systems (own table)

System Criteria	Apache Storm	Apache Flink	Apache Spark Streaming	Apache Samza
Written in	Java, Closure	Java, Scala	Scala, Java, Python	Scala, Java
Message Treatment	Stream(s)	Stream(s) and Batches	(Small) Batches	Stream(s)
Latency	Very Low	Very Low	High	Low
Throughput	Low	High	High	High
Message Processing	At-least-once Processing	Exactly-once Processing	Exactly-once Processing	At-least-once Processing
Main System Compo-nents	Nimbus (Master), ZooKeeper Nodes, Supervisors (Worker)	Job Manager (Master), Task Managers (Worker)	Cluster Manager (Master), Worker Nodes	YARN Resource Manager (Master), YARN Node Managers (Worker), Kafka Brokers

Furthermore, there are differences in how incoming data feeds are treated. As mentioned, Spark Streaming divides streams into batches of data, which are then further processed. That is contrary to the other systems, though Flink offers a batch processing as well.

Regarding latency and throughput, there are no benchmarks or measurements that compare all systems with each other. Hence, there are no numbers that would create a clear ranking for these quantifiable attributes. However, a few measurements exist for sub-sets of the compared systems [28], [7]. But since aspects like the latency can be highly dependent on the used algorithms and so on the strengths and weaknesses of each system, the results can differ between studies. Thus, the values displayed in table I can only be tendencies.

The kind of message processing assurance differs as mentioned among the systems. While Flink and Spark guarantee an exactly-once processing, Storm and Samza only assure that each message is processed at least once. According to [38], an option for exactly-once processing is also planned for Samza.

As mentioned beforehand, all systems make use of a master-worker pattern in their default setup with regard to the cluster architecture and do not use one single node type like, for example, Kafka. That can possibly create a single point of failure, the master node(s). Next to the master and worker nodes, Storm has the shown third layer in between, a ZooKeeper cluster. Contrary to the other systems, Samza is rather a mix of different exchangeable projects, whereby the actual cluster architecture is dependent on the used implementations for Samza's components.

V. Conclusions and Future Work

After analyzing and comparing the presented data stream processing systems, further research based on these results can include work in the area of benchmarks for such systems. As mentioned before, this topic is not fully developed and offers potential. Building on that, the creation of use case-driven recommendations for systems or concepts could be another interesting research area.

The mentioned benchmarking topic could thereby cover quantifiable aspects such as throughput and latency. These factors might also vary among systems depending on used algorithms. Moreover, a scalability analysis could be done. Particularly, the ratio between adding (worker) nodes and the resulting performance improvements is of peculiar interest.

As all compared systems more or less use a master-worker pattern, it might be interesting having a look at another concept. Additional domains that could be studied in greater detail are, for example, fault management, memory consumption and the used programming model for writing applications.

Regarding the near future, it will be interesting to observe towards which direction(s) the systems will develop. Additionally, one will see whether all projects will be continued next to each other or the development for one or more systems will stop. Twitter, for example, already announced Twitter Heron as its internal successor of Storm [43].

Summarizing, all systems have similarities, such as the fact that all systems run within a JVM, as well as distinctions in certain areas. Depending on the challenge that shall be tackled with a data stream processing system, there might be small advantages for one or another. A clear ranking can not be created based on the presented results, but the development of recommendations based on use cases or other aspects can be a next step for research.

References

[1] K. Ashton, "That internet of things thing," *RFiD Journal*, vol. 22, no. 7, pp. 97–114, 2009.

[2] S. Sarma, D. L. Brock, and K. Ashton, "The networked physical world," *Auto-ID Center White Paper MIT-AUTOID-WH-001*, 2000.

[3] L. Liu, C. Pu, and W. Tang, "Continual queries for internet scale event-driven information delivery," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, no. 4, pp. 610–628, Jul. 1999.

[4] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, Jun. 2003.

[5] L. Golab, K. G. Bijay, and M. T. Özsu, "Multi-query optimization of sliding window aggregates by schedule synchronization," in *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, 2006, pp. 844–845.

[6] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux, "Tristan: Real-time analytics on massive time series using sparse dictionary compression," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 291–300.

[7] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014, pp. 69–78.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 215–226.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Stanford InfoLab, Technical Report 2004-20, 2004.

[10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[11] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[12] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[13] V. Nguyen and R. Kirner, "Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, J. Koodziej, B. Di Martino, D. Talia, and K. Xiong, Eds. Springer International Publishing, 2013, vol. 8285, pp. 357–369.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076

[15] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156.

[16] D. Simoncelli, M. Dusi, F. Gringoli, and S. Niccolini, "Scaling out the performance of service monitoring applications with blockmon," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, M. Roughan and R. Chang, Eds. Springer Berlin Heidelberg, 2013, vol. 7799, pp. 253–255.

[17] P. Hintjens, *ZeroMQ: Messaging for Many Applications*, ser. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[19] S. Long, R. Rao, W. Miao, and X. Zhang, "An improved topology schedule algorithm for storm system," in *Computer Science and Applications: Proceedings of the 2014 Asia-Pacific Conference on Computer Science and Applications (CSAC 2014), Shanghai, China, 27-28 December 2014*. CRC Press, 2015, p. 187.

[20] "Spark - cluster mode overview," http://spark.apache.org/docs/latest/cluster-overview.html, accessed: 2015-08-11.

[21] I. Brigadir, D. Greene, P. Cunningham, and G. Sheridan, "Real time event monitoring with trident," in *RealStream: Real-World Challenges for Data Stream Mining workshop at European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2013), Prague, September 23th to 27th, 2013*, 2013.

[22] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[23] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[24] "Mirror of apache flink," https://github.com/apache/flink, accessed: 2015-08-20.

[25] K. Tzoumas, S. Ewen, and R. Metzger, "High-throughput, low-latency, and exactly-once stream processing with apache flink - the evolution of fault-tolerant streaming architectures and their performance," http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/, accessed: 2015-08-20.

[26] J. D. Bali, "Streaming graph analytics framework design," 2015.

[27] "Apache flink - configuration," https://ci.apache.org/projects/flink/flink-docs-master/setup/config.html, accessed: 2015-08-10.

[28] J. Galilee and Y. Zhou, "A study on implementing iterative algorithms using bigdata frameworks," http://sydney.edu.au/engineering/it/research/conversazione-2014/Galilee-Jack.pdf, accessed: 2015-08-20.

[29] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache spark perspective," *Global Journal of Computer Science and Technology*, vol. 15, no. 1, 2015.

[30] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[31] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438.

[32] "Mirror of apache spark," https://github.com/apache/spark, accessed: 2015-08-11.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[34] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[36] "Samza architecture," http://samza.apache.org/learn/documentation/0.9/introduction/architecture.html, accessed: 2015-08-17.

[37] "Mirror of apache samza," https://github.com/apache/samza, accessed: 2015-08-17.

[38] "Samza - storm," https://samza.apache.org/learn/documentation/0.9/comparisons/storm.html, accessed: 2015-08-20.

[39] "Hadoop," http://hadoop.apache.org, accessed: 2015-08-17.

[40] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[41] R. Pinilla and M. Gil, "Jvm: platform independent vs. performance dependent," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 2, pp. 44–56, 2003.

[42] R. A. Vivanco and N. J. Pizzi, "Scientific computing with java and c++: a case study using functional magnetic resonance neuroimages," *Software: Practice and Experience*, vol. 35, no. 3, pp. 237–254, 2005.

[43] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD*

International Conference on Management of Data, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250.