Scalable Distributed Stream Processing Systems

Jayapriya Surendran

Department of Computer Science, University of Minnesota, Twin Cities, MN – 55414
{suren009@umn.edu}

Abstract

As the amount of data generated grows exponentially, the need to process and analyze this data becomes imperative. Several systems have been developed in the recent days to provide real time stream processing capabilities. In this short survey, we analyze six such notable stream processing systems namely Apache Storm, Spark Streaming, Apache Samza, MillWheel, Timestream and Apache Flink.

1 INTRODUCTION

Due to the Internet expansion and massive usage of mobile phones, enormous amount of data is being generated every second and the need to analyze this data is growing significantly. Streaming data analysis is inevitable for applications like surveillance and intelligence systems, fraud detection, e-commerce, risk management systems, network monitoring systems, market data management systems and transaction cost analysis systems.

Several big data processing systems are available to facilitate data analysis. Some data needs to be processed on regular intervals (hourly/nightly/weekly). Such data can be better processed by batch processing systems like MapReduce. But some data are most valuable at the time it arrives. There is no point in detecting fraud after the burglar has absconded or detecting a potential buyer after the user leaves your e-commerce site. This category of data needs to be processed in real time. Real time stream processing systems have been developed to process data from log streams, click streams, TCP streams, event streams etc in real time to get meaningful results.

Processing and analyzing hundreds of Megabytes of data every second in real time is a very challenging task. Luckily, many distributed real time stream processing frameworks have been developed in the recent times, which can process large amounts of data considerably fast. But the available open source stream processing systems are quite young and no single system provides a complete solution for all use cases. New problems in this area include: *How a stream processor's state should be managed? Whether or not a stream should be buffered remotely on disk? What to do when duplicate messages are received or messages are lost?* In this survey, we try to provide an overview of various real time stream-processing systems and also analyze their high level features with other systems.

The rest of the paper is organized as follows: Section 2 discusses the core technologies for real time stream processing systems. Section 3 gives a qualitative comparison of the six real time stream processing systems mentioned above. In Section 4, we explore languages supported by various stream processing systems. Finally, Section 5 concludes the paper.

2 CORE TECHNOLOGIES FOR STREAM PROCESSING SYSTEMS

In this section, we shall introduce some techniques and concepts that are important for efficient processing of data streams. These are the basics on which the performances of stream processing systems heavily rely.

2.1 Ordering and Guarantees

Ordering guarantees that the input is processed in the same order in which it is fed into the stream processing system. Also these systems provide three types of guarantees for data processing as follows

- At most once processing
- At least once processing
- Exactly once processing

MillWheel [7] framework performs the following steps upon the receipt of input for computation to guarantee exactly once delivery

- The record is checked against de-duplication data from previous deliveries and duplicates are discarded.
- User code is run for the input record, possibly resulting in pending changes to timers, state, and productions.
- Pending changes are committed to the backing store.
- ACK messages are sent back to the sender.
- Pending downstream productions are sent.

Spark Streaming [5] guarantees ordered processing of RDDs in one DStream. Since each RDD is processed in parallel, order cannot be guaranteed within the RDD.

TimeStream [8] provides strong exactly-once semantics based on lightweight dependency tracking. It also provides deterministic ordering (when needed) that preserves a well-defined ordering of inputs to each computation.

Apache Storm [3] allows the user to choose the level of guarantee (at most once, at least once and exactly once) with which the messages should be processed. Exactly once delivery is achieved using Trident API

2.2 State Management

Stream processors usually need to maintain some states as they process messages. Different frameworks have different approaches to handling such state, and resolve failure as well. State information is usually stored in-memory and periodically updated to persistent storage.

Spark Streaming [5] periodically writes intermediate data of stateful operations into the HDFS.

Apache Flink [9] takes periodic snapshots of the entire state of the topology and stores them in distributed store (HDFS or in-memory file system like Tachyon). Check pointing happens in the background without pausing the stream. It uses Chandy and Lamport's algorithm for consistent asynchronous distributed snapshots. This is similar to micro batching, in which all computations between two checkpoints either succeed or fail atomically as a whole.

Flink supports both asynchronous and incremental check pointing. It has three operator states namely

- *User-define state*: User-defined state maintenance within operator through Java/Scala objects. No backup and recovery for these states

- *Managed state*: special interface for user defined operations that can be backed up and restored during recovery
- *Windowed state*: maintains state of windowing operators

Apache Samza [6] takes a different approach for state management. Rather than using a durable remote database, each Samza task includes an embedded key-value store, located on the same machine. Changes to this key-value store are replication to other nodes in the cluster, so that if one machine dies, the state of the tasks it was running can be restored from another node/machine.

Apache Storm [3] offers automatic state management as part of higher level Trident API. It keeps the state in memory and periodically updates it to a remote database for durability, so the cost of remote database call is amortized over several processed tuples.

2.3 Partitioning and Parallelism

Stream processing framework has to partition the incoming data stream into independent executable tasks so that the data can be distributed across multiple nodes in a cluster for parallel computation.

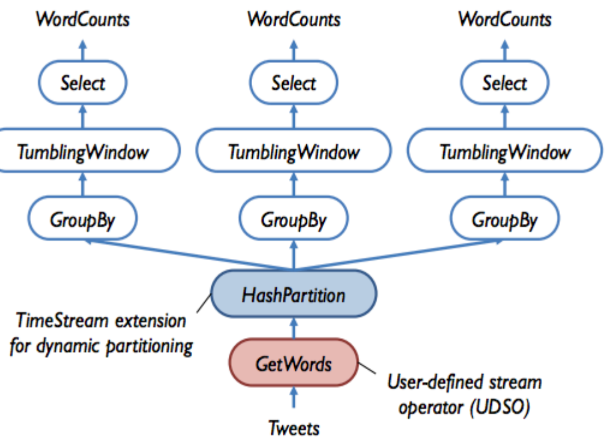

Figure 1: TimeStream DAG for continuous word count query

In **TimeStream [8]** new re-partitioning operator HashPartition<K,T> is used to repartition streams for parallel execution.

Apache Storm's [3] parallelism model splits processing into independent tasks that can run in parallel. Resource allocation is independent of the number of tasks. It also supports *dynamic rebalancing*, where threads or processes can be added to a topology without re-starting the topology or cluster.

The biggest difference between Storm and **Apache Samza [6]** is that Samza uses single threaded processes (containers) where as Storm uses one thread per task by default.

In **Apache Flink [9]** parallelism can be set on each operator. Each vertex can have one or more operator. This concept is similar to that of MapReduce, Tez and Spark.

Spark Streaming [5] achieves parallelism by splitting the job into small tasks and sending them to executors. Spark has two types of parallelism.

- Parallelism in receiving the stream and
- Parallelism in processing the stream.

2.4 Buffering and Latency

Real time stream processing systems have special mechanisms to reduce the communication latency between different nodes in the cluster.

MillWheel [7] stores the snapshots of checkpoints in local cache. It periodically sends cached data to persistent key-value store like Spanner or BigTable.

In **TimeStream [8]** each vertex has output buffer as a caching mechanism. If downstream vertex requests old data it can avoid re-computation. It also does asynchronous state check pointing on KV store. Because of this query, coordinator may see stale view of progress.

Apache Storm [3] uses ZeroMQ [15] for communication between bolts, which allows low latency transmission of tuples. On the flip side, when a bolt is trying to send messages using ZeroMQ, and the consumer can't read them fast enough, the ZeroMQ [15] buffer in the producer's process begins to fill up with messages. If this buffer grows too much, the topology's processing timeout may be reached, which causes messages to be re-emitted at the spout and makes the problem worse by adding even more messages to the buffer.

In **Apache Flink [9]** buffering is done to align barriers. Each operator buffers the input streams that emits data faster so as to align the barrier with the slow input stream. Operators also maintain output buffers whose contents are emitted after certain timeout or when full.

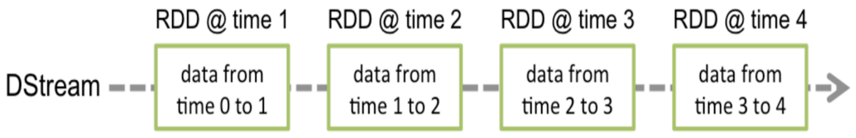

Figure 2: Spark Discretized Streams (DStreams)

Spark Streaming [5] is a sequence of small batch processes. If the processing is slower than receiving, the data will be queued as DStreams in memory and the queue will keep increasing. To run a healthy Spark application, we have to tune the system in such a way that receiving speed is equal to the processing speed.

2.5 Fault-tolerance

Real time systems should be resilient to failure and hence fault-tolerance mechanisms are incorporated into stream processing systems.

Apache Samza [6] and **Spark Streaming [5]** usually experience two types of failure.

- Worker node failure in Spark Streaming (similar to container failure in Samza)
- Driver node failure (equivalent to application manager AM failure in Samza)

When a worker node fails in Spark Streaming, the Cluster Manager will restart it. When a container fails in Samza, the application manager will work with YARN to restart a new container. When a driver node fails in Spark Streaming, YARN/Mesos/Spark Standalone will automatically restart the driver node.

Apache Flink [9] guarantees failure recovery/tolerance through distributed consistent snapshots. Re-computation based failure recovery is used to achieve strong exactly-once semantics. On failure, Flink restores the latest snapshot from durable storage and rewinds the stream source to the point when snapshot was taken and replaying the stream.

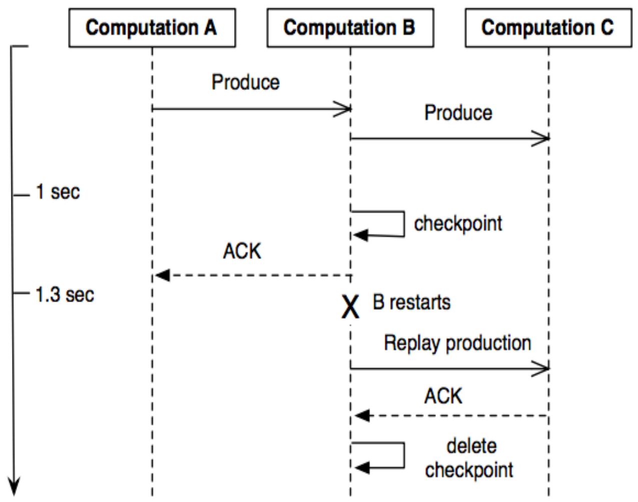

Figure 3: MillWheel Check pointing mechanism

MillWheel [7] guarantees Fault tolerance by sending ACK messages, implementing Strong and weak productions and by caching snapshot of the system in local cache until it gets ACK message from the receiver. Also Low watermark concept puts an upper bound on the value of the incoming timestamps.

2.6 Deployment and Execution

A **Storm [3]** cluster is composed of a set of nodes running a Supervisor daemon. The supervisor daemons talk to a single master node running a daemon called Nimbus. The nimbus daemon is responsible for assigning work and managing resources in the cluster.

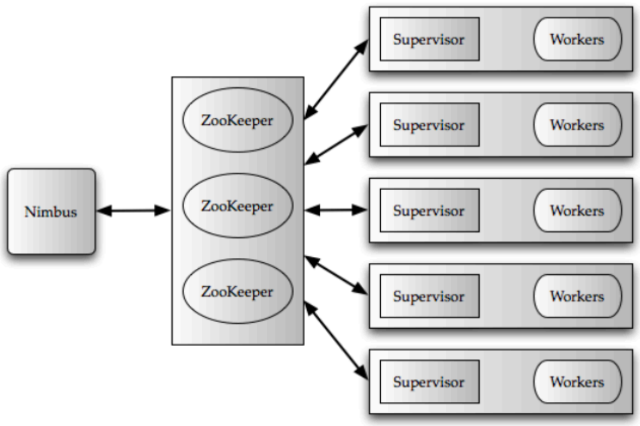

Figure 4: High Level Storm Deployment

In **TimeStream [8]** Cluster Manager is used for resource management and scheduling vertices on slave nodes. Query coordinator persists metadata information for fail-over. Each slave node has fixed number of vertex processes and each vertex process can run one or more tasks.
Apache Samza [6] leverages YARN to deploy user code, and execute it in a distributed environment.

Apache Flink [9] - Job Manager is responsible for scheduling and resource management, which is a single point of failure. Task Manager is responsible for task execution and data exchange. Flink can also run on top of YARN.

Spark Streaming [5] has a SparkContext object to talk with cluster managers, which then allocate resources for the application. Currently Spark Streaming supports three types of Cluster managers: Spark standalone, Mesos and YARN.

3 QUALITATIVE COMPARISONS

In this section, we summarize the stream processing systems elaborated in this paper in terms of Ordering and Guarantees, State Management, Partitioning and Parallelism, Deployment and Execution, Buffering and Latency, Language Support, Maturity, Data model and Fault Tolerance in Table 1.

	MillWheel	**TimeStream**	**Apache Storm**	**Apache Samza**	**Spark Streaming**	**Apache Flink**
Organization	Google	Microsoft	Twitter	LinkedIn	UC Berkeley	Data Artisans
Written in	N/A	.NET, C#	Java,Clojure	Java,Scala	Java,Scala	Java,Scala
Open Source	No	No	Yes	Yes	Yes	Yes
License	Proprietary	Proprietary	Apache	Apache	Apache	Apache
Data Model (Finest data granularity)	Keys	Events	Tuples. Pluggable serialization	Pluggable data model and serialization	DStream (sequence of RDDs)	Records
Processing Unit	Computation	Vertex	Bolt	Processor	Executor/Worker	Operator
Data Representation	<key,value, timestamp>	<key,value>	<key,value>	<key,value>	<key,value>	<key,value>
Reliability Model/Guarantee	Exactly once execution	Exactly once execution	Exactly once/ atleast once/ atmost once	Atleast once execution	Exactly once execution	Exactly once execution
Data Ordering	Gracefully handles out of order execution with timestamp and Low watermark	Deterministic ordering can be enforced	Out of order execution is possible	Ordering guaranteed via Kafka	Order not guaranteed within RDD	Ordering guaranteed
Partitioning/ Parallelism	N/A	Hash Partition	One thread per task.	Single threaded processes (containers). Can run multiple tasks	DStream parallelism (within RDD)	Can be set on each operator. Each vertex can have more than 1operator
State Management	Persistent storage (Big Table, Spanner)	Maintains locally at each vertex. Periodically updates key value store	In- memory. Periodically updates checkpoints to database	Maintains locally with task	In-memory. Periodically updates to HDFS	Persistent Storage
Buffering and Latency	Stores snapshots in local cache.	Asynchronous state checkpointing in key-value store	Uses ZeroMQ for low latency communication	Persists to an output stream	Caches in memory	Input/output buffers.
Fault Tolerance	Strong & weak productions/ ACK messages	Resilient re-substitution	ACK messages	YARN's default mechanism	Lineage Graph	Distributed consistent snapshots
Dynamic Reconfiguration/ Rebalancing	No	Yes	Yes	No	No	No
Deployment and Execution	N/A	Cluster Manager	Nimbus/YARN/ Mesos	YARN	Cluster Manager/ YARN,Mesos	Job Manager/YARN
Language Support	N/A	LINQ/.NET/C#	Java, Clojure, Scala, Python, Ruby	Java,Scala	Java,Scala, Python	Java,Scala, Python
Isolation	Process level	N/A	UNIX process level isolation	YARN container isolation	YARN/Mesos container isolation	YARN container isolation

Table 1 Comparison of Real time Stream Processing Systems

4 LANGUAGE SUPPORT

Apache Storm [3] is written in Java and Clojure but has good support for non-JVM languages. It follows a model similar to MapReduce streaming: a non-JVM task is launched in a separate process, data is sent to it's stdin, and output is read from it's stdout.
Apache Flink [9] provides Java, Scala and Python support
TimeStream [8] requires Microsoft StreamInsight on top of .Net platform. Queries are written in LINQ with a .Net language such as C#.
Apache Samza [6] is written using Scala and Java and currently provides support for JVM languages.
Spark Streaming [5] is written using Java and Scala and provides Java, Scala and Python APIs.

5 CONCLUSIONS

In this survey, we identified key attributes of distributed stream processing systems and evaluated several leading distributed stream processing systems based on those attributes. None of these models have taken a clear lead in the stream processing space and it is too early to tell about their success. With the growth of the Internet of Things in the coming years, the demand for large scale distributed stream processing will also increase and it is important to find formal stream processing models to lay a foundation to build standard data analytics algorithms on such systems.

6 REFERENCES

[1] Apache hadoop. http://hadoop.apache.org, 2012.
[2] J. Dean and S. Ghemawat. *Mapreduce: simplified data processing on large clusters*. Commun. ACM, 51:107113, Jan. 2008.
[3] N. Marz. Twitter storm. https://github.com/nathanmarz/storm/wiki, 2012.
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of 9th USENIX conference on Networked Systems Design and Implementation, 2011.
[5] 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 Computing, pages 10–10. USENIX Association, 2012.
[6] Apache Samza. http://samza.apache.org
[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle: *MillWheel: Fault-Tolerant Stream Processing at Internet Scale*. PVLDB 6(11): 1033-1044 (2013)
[8] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, Zheng Zhang: *TimeStream: Reliable Stream Computing in the Cloud*. Eurosys'13: ACM 978-1-4503-1994-2/13/04
[9] Paris Carbone, Gyula Fra, Stephan Ewen, Seif Haridi, Kostas Tzoumas: Lightweight Asynchronous Snapshots for Distributed Dataflows.
[10] Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, Roman Schindlauer: *The Extensibility Framework in Microsoft StreamInsight*. ICDE 2011: 1242-1253
[11] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, Jimmy Lin: *Summingbird: A Framework for Integrating Batch and Online MapReduce Computations*. PVLDB 7(13): 1441-1451 (2014)
[12] Spark Streaming. https://spark.apache.org/streaming/
[13] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Ryaboy: Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy *Storm@Twitter*. SIGMOD 2014: 147- 156
[14] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley,

Sanjay Radia, Benjamin Reed, Eric Baldeschwieler: *Apache Hadoop YARN: Yet Another Resource Negotiator*. SoCC 2013: 5

[15] ZeroMQ: http://zeromq.org. Retrieved December 1, 2014

[16] Jay Kreps, Neha Narkhede, and Jun Rao. *Kafka: a distributed messaging system for log processing.* SIGMOD Workshop on Networking Meets Databases, 2011.

[17] Kestrel: A simple, distributed message queue system. http://robey.github.com/kestrel

[18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. *Mesos: a platform for fine-grained resource sharing in the data center*. In NSDI, 2011.

[19] Apache Zookeeper. http://zookeeper.apache.org/

[20] Trident. https://github.com/nathanmarz/storm/wiki/Tridenttutorial.

[21] Hunt, P., Konar, M., Junqueira, F. P., And Reed, B. *Zookeeper: Wait-free coordination for internet-scale systems*. In USENIXATC, 2010

[22] Lamport, L. *Paxos made simple, fast, and byzantine*. In OPODIS, 2002.

[23] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, Siddarth Taneja: *Twitter Heron: Stream Processing at Scale*. SIGMOD Conference 2015: 239-250

[24] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, Matei Zaharia: *Scaling Spark in the Real World: Performance and Usability*. PVLDB 8(12): 1840-1851 (2015)

[25] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia: *Spark SQL: Relational Data Processing in Spark*. SIGMOD Conference 2015: 1383-1394