Scientific Journal of Impact Factor(SJIF): 3.134
e-ISSN(O): 2348-4470
p-ISSN(P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 2,Issue 5, May -2015

A Survey of Stream Processing Frameworks for Big Data

Mansi Shah[1], Vatika Tayal[2]

[1]M. Tech. Scholar, Computer Science and Engineering Department, N.S.I.T, Jetalpur, Gujarat
[2]Assistant Professor, Computer Science and Engineering Department, N.S.I.T, Jetalpur, Gujarat

Abstract — *In recent years due to the acceleration in IoT (Internet-of-Things) and M2M (Machine-to-Machine) communications streams are everywhere. Twitter streams, log streams, TCP streams click streams and event streams are some good examples. Big data streaming applications need to process and analyze information in real-time. The Map/Reduce model and its open source implementation Hadoop designed as a high fault-tolerant system for batch processing and high throughput jobs. However, the Map/Reduce framework is not suitable real-time streaming applications that require very low latency of response. Owing to the high demand for processing non-batch jobs such as real-time and streaming jobs several big data frameworks have been developed or under developing. This paper presents a survey of open source frameworks that support big data stream processing.*

Keywords- *big data, stream processing, architectures*

I. INTRODUCTION

With the acceleration in IoT (Internet-of-Things) and M2M (Machine-to-Machine) communications there is a class of emerging stream data applications such as telematics, sensor- based monitoring, network monitoring, fraud detection, traffic estimation, stock trading and so on where tremendous volume of data generated with velocity in external environments are pushed to servers for real-time processing. The data generated by these applications can be viewed as an unbounded sequence of events where most of the data is valuable at its time of arrival. For example, Credit card fraud analytics or sensor-based network fault prediction to predict if a given transaction is a fraud or if the network is developing a fault need to process real-time data stream on the fly at its arrival. If decisions such as these are not taken in real-time, the chance to alleviate the damage is lost. Big data streaming applications have a high volume, high velocity and complex data types. However, the standard MapReduce model and its implementations like Hadoop, is completely focused on batch processing and handle only the volume and variety of the data but not the velocity part of it. That is, all input data must be completely available in the input store before any computation is started and the output results are available only when the entire computation is done. In contrast to these batch properties, for stream applications input data is not available completely in the beginning and arrives constantly. Also, the input data must be processed without being totally stored. These new demands for large-scale stream processing require systems that are more elaborate, agile and sophisticated than the recently available Map/Reduce solutions like the Hadoop framework. This paper surveys the frameworks that can handle big data stream processing [1].

II. BIG DATA STREAM PROCESSING FRAMEWORKS

We now introduce real-time Big Data frameworks that are widely designed for real-time stream data analytics.

2.1. Storm

Storm is an open source distributed and fault-tolerant real-time framework for processing unbounded streaming data developed by Twitter. It guarantees all the data will be processed and is easy to set up and operate. Storm is fast that a benchmark clocked it at over a million tuples processed per second per node. Therefore, it has many use cases, such as real-time analytics, interactive operation system, on-line machine learning, continuous computation, distributed RPC, and so on [2].

2.1.1. Storm Topology

To implement real-time stream processing on Storm, users need to create different topologies as shown in Figure 1. A topology is arranged as directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Spouts are the starting points in the graph, which act as source of streams. Bolts process the input streams that are piped into it and outputs new streams. Each node in the topology contains processing logic and executes in parallel. The links between nodes indicate how the data should be processed by nodes. Spouts and bolts can be written in different programming languages like Python, Java or Clojure [1].

@IJAERD-2015, All rights Reserved

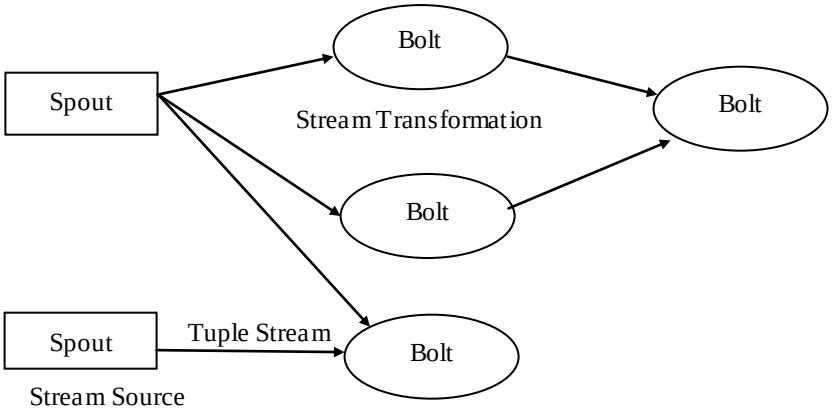

Figure 1. Storm Topology

2.1.2. Storm Architecture

The high level Storm architecture is shown in Figure 2. A Storm cluster consists of three sets of working nodes and they are Nimbus, Zookeeper and Supervisor nodes. Nimbus acts as the master node and plays the role of JobTracker in Hadoop. It is responsible for distributing code across the Storm cluster, scheduling tasks to worker nodes and coordinating the execution of the whole system. Supervisors act as worker nodes and play the role of TaskTracker in Hadoop. Clients describe the topology as Thrift object and submits to Nimbus. Nimbus distributes the code to the workers for execution, keeps a track of the progress of the workers and handles node failures. The actual work is done by workers which receives instructions from Nimbus and spawns workers based on it. Each worker process runs a Java Virtual Machine (JVM), in which it runs one or more executors. Executors are composed of one or more tasks. The actual work for a spout or a bolt is done in the task. The Supervisors contact Nimbus with a periodic heartbeat protocol, advertising the currently running topologies and any vacancies that are available to run more topologies. Zookeeper plays an important role in coordinating Nimbus and Supervisor nodes. Furthermore, it records all states of Nimbus and Supervisors on the local disk for resilience. If Nimbus node fails, the workers can still continue to make progress. Also, the Supervisors restart the workers if they fail [3].

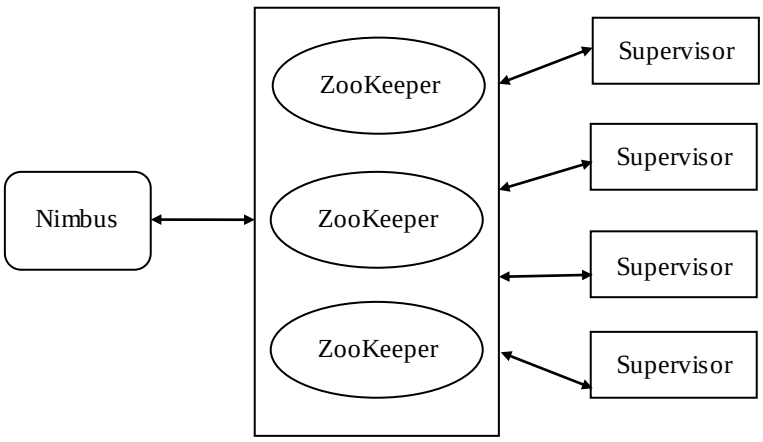

Figure 2. Storm Architecture

2.1.3. Fault Tolerance

In Storm, Nimbus handles the node failures. The Nimbus and Supervisor daemons are designed to be stateless and fail-fast. The Supervisor nodes periodically send heartbeats to Nimbus. If heartbeats are not received by Nimbus timely, it assumes that the supervisor is no longer active. Node failure and message failure are two orthogonal events as message failure can result from software bugs or intermittent network failures. Due to this handling of failed messages and moving workers to other nodes in the event of node failures are done in two different ways without any correlation between the two. This design is what makes the system more robust to failures [3].

@IJAERD-2014, All rights Reserved

2.2. Apache S4

S4 (Simple Scalable Streaming System) is a general-purpose, fault-tolerant, distributed, decentralized, scalable, event-driven, modular computing platform for processing continuous unbounded streams of data. It was initially released by Yahoo in 2010 and has become an Apache project since 2011. The processing model is inspired by MapReduce and employs the Actors model for computations. Processing elements (PE's) are written using Java programming language. PE's are assembled into applications using the Spring Framework based on XML configuration [2].

2.2.1. S4 Architecture

S4 is a message-passing system. Figure 3 shows the S4 architecture. S4 provides Client Adapter which allows third-party client to send and receive events to S4 cluster. Client Stub component communicates with the clients using TCP / IP protocol functions. The two core components of the S4 framework are Events and Processing elements. Events are the only mode of communication between the Processing Elements. The external clients act as the source of data which is submitted to the client adapter. Also, the external clients can receive events from S4 cluster through the adapter via the Communication Layer. The adapter converts the incoming input data into events which are then sent to the S4 cluster [5].

Processing Elements (PEs) are the basic computational units that identify the events with the help stream names. Each runtime instance of a PE is uniquely identified by using its functionality as defined by class and configuration, the types of events consumed by the PE, the keyed attribute in those events and the value of the keyed attribute in the event it consumes. A new PE is instantiated for each unique value of the key attribute. .Every PE consumes data events routed towards it on the basis of keys and either produces one or more events to be consumed by other PE's or publishes the results to an external database or consumer [4].

Processing Nodes (PNs) act as the logical hosts to PEs. They are responsible for listening to incoming events, executing operations on the incoming events, dispatching events and generating output results. S4 initially routes every event to PNs based on a hash function of the values of all known keyed attributes in that event. When the event reaches the appropriate PN, an event listener in the PN sends the incoming event to the processing element container (PEC) which invokes the appropriate PEs in the proper order [4].

The communication layer is responsible for cluster management, mapping physical nodes to logical nodes and automatic failure handling. This layer uses distributed service Zookeeper to help coordinate between nodes in an S4 cluster [5].

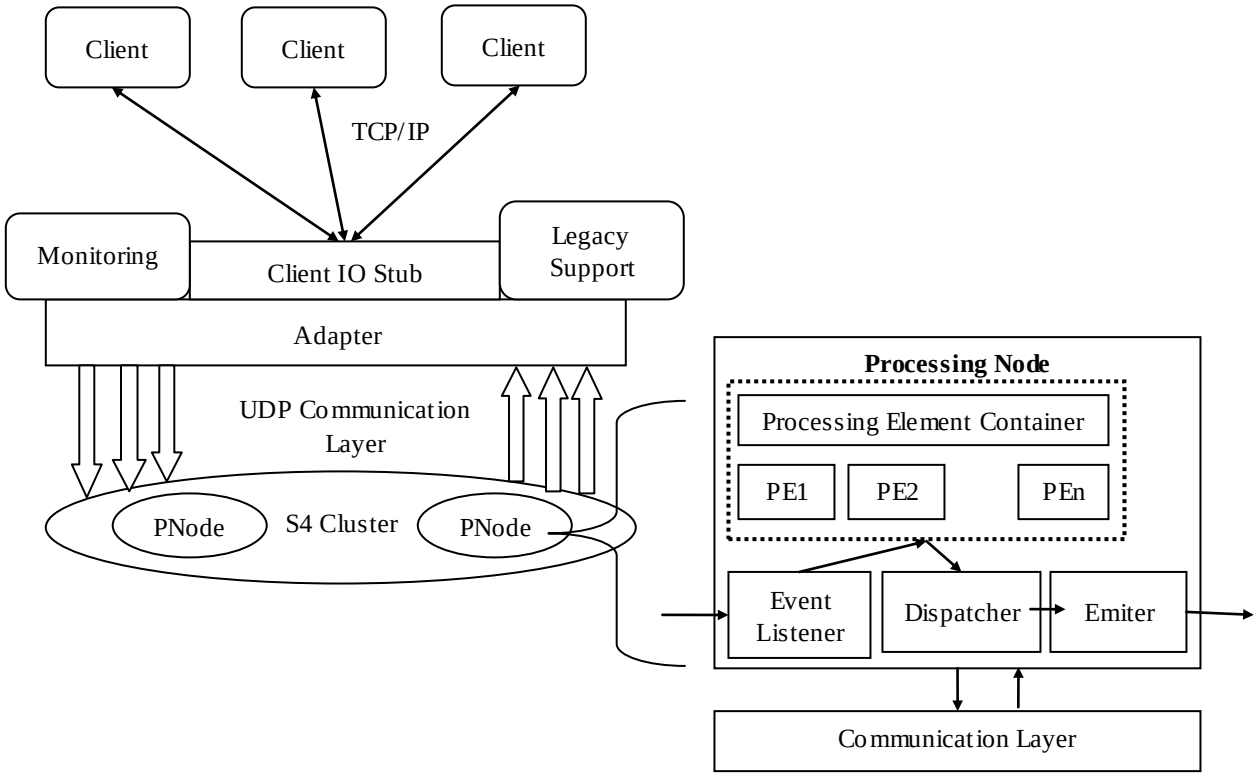

Figure 3. S4 Architecture

2.2.2. Fault Tolerance

In the presence of sudden node failures, S4 automatically detects the failure using Zookeeper and distributes the tasks assigned to the failed node to other nodes. Snapshots of the state of the processing nodes are saved time to time and these are used to create a new instance of a Processing Node when one crashes. S4 doesn't guarantee message delivery in case of node failures and uses State recovery for recovering from failures. Events sent after the last checkpoint and before the recovery are lost. Also, S4 uses the push model of events in the system so events can be lost due to high load. Because of these two reasons State recovery is very important for long running machine learning programs [4] [6].

2.3. Apache Spark Streaming

Spark Streaming is an extension of the core Spark API that enables high-throughput, in memory, scalable, fault-tolerant stateful stream processing of live data streams. Data can be inserted from many sources like Kafka, Flume, ZeroMQ, Kinesis, Akka Actor or TCP sockets and can be processed using algorithms expressed with high-level functions like map, reduce, join and so on. Finally, processed data can be stored to file systems like HDFS, databases and live dashboards [7].

2.3.1. Spark Streaming Architecture

Spark Streaming treats each streaming job as a series of deterministic batch jobs of small time intervals. It provides an abstraction called Discretized Streams (D-Streams), which represents a continuous stream of data. Internally, each D-Stream is represented by a continuous series of resilient distributed datasets (RDD's), which is Spark's abstraction of an immutable, fault-tolerant and distributed datasets. Each RDD in a D-Stream can be acted on by deterministic transformations and contains data from a particular interval. The batch processing engine Spark is used to process each batch of data. Spark Streaming will receive the live input data stream, divide it into batches of one second and store them in Spark's memory as RDDs.

Once the time interval completes, the dataset of the corresponding interval is processed via deterministic parallel operations, such as map, reduce, reduceByKey and groupBy, to produce new datasets representing either program outputs or intermediate state. In the former case, the results may be stored in an external file system or storage device. In the latter case, the intermediate state is stored as resilient distributed datasets (RDDs) which may then be processed along with the next batch of input data to produce a new dataset of updated intermediate states [8].

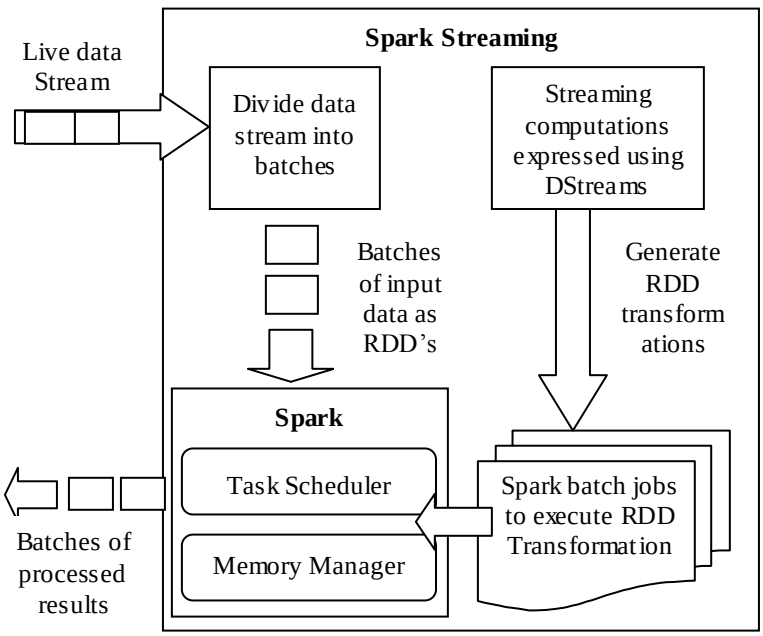

Figure 4. Spark Streaming Architecture

2.3.2 Fault Tolerance

Fault tolerance is essential for stream processing. To recover from failures, both DStreams and RDDs track of the deterministic operations used to build them called the lineage graph, and reruns these operations on base data to rebuild lost partitions. When a node fails, it reconstructs the RDD partitions that were on it by re-running the operations that built them from the original input data reliably stored in the cluster [8].

2.4. Apache Samza

@IJAERD-2014, All rights Reserved

Apache Samza is an open source and distributed real-time computational framework for processing streaming data. It was initially by LinkedIn and has become an Apache incubator project since 2013. Kafka is used for messaging and YARN to provide fault tolerance, security, processor isolation, cluster management and resource management [9].

2.4.1. Processing Model

The basic building blocks of a Samza application are: Streams and Jobs.A stream is composed of an immutable collection of messages of same type or category. A stream can be read by any number of consumers and messages can be added to or deleted from a stream. Each stream is broken into one or more partitions to scale the system to handle large amount of data. The sequence of messages within each partition is totally ordered.

A job is a code that performs a logical transformation on a set of input streams and produce output streams. In order to increase the throughput of the processor, each job is broken into smaller units of execution called Tasks. A task act on one partition of a message stream and produce a message stream. Each task can consume data from multiple partitions from different input streams. Tasks can operate independently as there is no defined ordering of messages across the partitions [9].

2.4.2. Samza Architecture

Samza architecture consists of three layers: streaming layer, execution layer and processing layer. It provides support for the three layers using Apache Kafka, Apache Yarn and Samza API respectively. Kafka is used for the distributed message brokering with persistence for message streams. Yarn is used for the distributed resource allocation, scheduling and task coordination across machines. Samza API responsible for creating, processing stream tasks on a cluster. In Kafka, a stream is called a topic. A topic is partitioned using a partitioning scheme and replicated across multiple machines called brokers. The partitioning of a stream is done on the basis of the key associated with the messages in the stream. The messages with the same key belong to the same partition. In order to achieve distributed partitioning of streams, Samza uses the topic partitioning of Kafka. Kafka brokers provide the input and output for the Samza StreamTasks. YARN has three important blocks: a ResourceManager (RM), a NodeManager (NM), and an ApplicationMaster (AM). To start a new job, the Samza client talks to the RM. To allocate space on the cluster for Samza's ApplicationMaster the YARN RM talks to a YARN NM. NM starts the Samza AM after space allocation. Once the Samza AM starts, it tells the YARN RM for one or more YARN containers to run SamzaContainers and the processing code runs inside these containers. The RM talks to NM and allocates space for the containers. After the space allocation, the NM's start the containers.Kafka stores all the messages in the file system and doesn't delete them for a configured amount of time which allows the tasks to consume messages at arbitrary points if they need to [10].

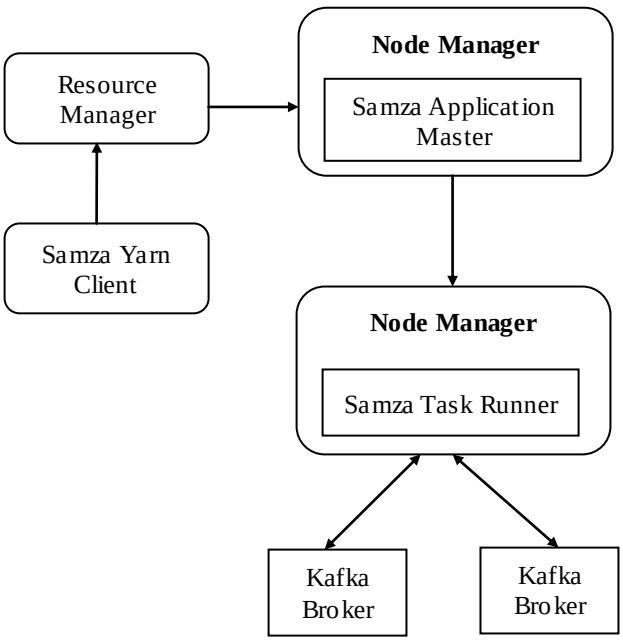

Figure 5. Samza Architecture

2.4.3. Fault Tolerance

Samza provides fault tolerance by restarting containers that fail and continue processing of the stream. Samza uses "checkpoints" to restart from the same offset. For each input stream partition that a task consumes, the Samza container periodically checkpoints the current offset. When the container restarts after a failure, it inspects the most recent

@IJAERD-2014, All rights Reserved

checkpoint and starts consuming messages from the checkpointed offsets. This guarantees at-least-once delivery and processing of messages. However, if a broker node fails messages persisted in the file system are lost and cannot be recovered [9].

III. COMPARISION OF STREAM PROCESSING FRAMEWORKS

Table 1. Comparision of Stream Processing Frameworks [11][12]

Specification	Storm	Spark Streaming	S4	Samza
Data Pipeline	Pull based	Pull based	Push based	Pull based
Delivery Semantics	At Least Once Exactly-Once with Trident	Exactly Once	Not guaranteed delivery	At Least Once
State Management	Stateless	Stateful	Stateless	Stateful
Data Querying	Trident	Spark SQL	None	None
Latency	Sub-second	Seconds Depending on batch size	Seconds	Sub-second
Language Support	Java, Clojure, Ruby, Python, Javascript, Perl	Scala, Java, Python	Java	Scala, Java

IV. CONCLUSION

With the exponential growth in IoT and M2M communication, dealing with large amounts of streaming data is posing new challenges. MapReduce-based solutions like Hadoop is suitable for batch processing jobs that process large amount of data over a long time. This high latency response makes it unsuitable for recent demands of real-time stream processing. Hence, real-time stream processing frameworks have emerged recently which have shown the advantage in handling continuous data streams and provide stream data analytics. In this paper, we have made the survey of open-source stream processing frameworks, including Storm, Spark Streaming, S4 and Samza. We have also compared the stream systems from the perspectives of data pipeline, delivery semantics, state management, data querying, latency and language support. Each framework has its drawbacks and advantages. With the increase in streaming data, it would be of a great value to come up with an efficient framework for gathering, processing and analyzing big data in a near real-time.

REFERENCES

[1] Dibyendu Bhattacharya, Manidipa Mitra, "Analytics on Big Fast Data Using Real Time Stream Data Processing Architecture", EMC Proven Professional Knowledge Sharing, 2013.

[2] C.L. Philip Chen, Chun-Yang Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data", Information Sciences, vol. 275, pp. 314–347, August 2014.

[3] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni,Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Storm @Twitter, SIGMOD'14, pp. 22–27, June 2014

[4] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari, "S4: Distributed Stream Computing Platform", Data Mining Workshops (ICDMW), 2010 IEEE International Conference, pp. 170-177, Dec. 2010.

[5] Jagmohan Chauhan, Shaiful Alam Chowdhury and Dwight Makaroff, "Performance Evaluation of Yahoo! S4: A First Look", P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference, pp. 58-65, Nov. 2012.

[6] S4 distributed stream computing platform, http://incubator.apache.org/s4/, 2015.

[7] Spark Streaming Programming Guide, https://spark.apache.org/docs/latest/streaming-programming-guide.html, 2015.

@IJAERD-2014, All rights Reserved

[8] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica, "Discretized streams: fault-tolerant streaming computation at scale", SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 423-438, Nov. 2013.

[9] Apache Samza: LinkedIn's Stream Processing engine, https://engineering.linkedin.com/samza/apache-samza-linkedin-stream-processing-engine, 2015.

[10] Samza, http://samza.apache.org/learn/documentation/latest/introduction/architecture.html, 2015.

[11] Streaming Big Data: Storm, Spark and Samza, https://tsicilian.wordpress.com/2015/02/16/streaming-big-data-storm-spark-and-samza/, 2015.

[12] Supun Kamburugamuve, "Survey of Distributed Stream Processing for Large Stream Sources", Technical Report, 2013. Available at http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf.

@IJAERD-2014, All rights Reserved