IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 28, 2019, accepted September 25, 2019, date of publication October 10, 2019, date of current version November 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2946884

A Survey of Distributed Data Stream Processing Frameworks

HARUNA ISAH[1], (Member, IEEE), TARIQ ABUGHOFA[1], SAZIA MAHFUZ[1], DHARMITHA AJERLA[1], FARHANA ZULKERNINE[1], (Member, IEEE), AND SHAHZAD KHAN[2]

[1]School of Computing, Queen's University, Kingston, ON K7L 2N8, Canada
[2]Gnowit, Ottawa, ON K1Y 2C5, Canada

Corresponding author: Haruna Isah (isah@cs.queensu.ca)

This work was supported by SOSCIP under the TalentEdge Post-doctoral Fellowship (OCE Project Number: 28379) and Cloud Project (SPEDF1-042) entitled "A Multilevel Streaming Data Analytics Infrastructure for Predictive Analytics". Computations were performed on the SOSCIP Consortium's Cloud Data Analytics and Large Memory System computing platforms. SOSCIP is funded by the Federal Economic Development Agency of Southern Ontario, the Province of Ontario, IBM Canada Ltd., Ontario Centres of Excellence, Mitacs and Ontario academic member institutions.

ABSTRACT Big data processing systems are evolving to be more stream oriented where each data record is processed as it arrives by distributed and low-latency computational frameworks on a continuous basis. As the stream processing technology matures and more organizations invest in digital transformations, new applications of stream analytics will be identified and implemented across a wide spectrum of industries. One of the challenges in developing a streaming analytics infrastructure is the difficulty in selecting the right stream processing framework for the different use cases. With a view to addressing this issue, in this paper we present a taxonomy, a comparative study of distributed data stream processing and analytics frameworks, and a critical review of representative open source (Storm, Spark Streaming, Flink, Kafka Streams) and commercial (IBM Streams) distributed data stream processing frameworks. The study also reports our ongoing study on a multilevel streaming analytics architecture that can serve as a guide for organizations and individuals planning to implement a real-time data stream processing and analytics framework.

INDEX TERMS Dataflow architectures, data stream architectures, distributed processing systems comparison, survey, taxonomy.

I. INTRODUCTION

The ability to handle and process continuous data streams is becoming an essential part of building a data-driven organization. Data streams are sequences of unbounded tuples generated continuously in time [1]. Unbounded and global datasets such as Web logs, mobile usage statistics, and sensor networks are increasingly becoming common in day-to-day businesses [2]. Streaming data is also increasingly getting important for social media platforms such as Facebook, Twitter, and LinkedIn [3].

Unlike traditional batch processing which involves processing of static data, streaming or online processing involves processing dynamic or continuous data [4]. Extracting meaningful and timely insights from unbounded data is very challenging. Recent demands for scalable and low latency analytics tools [5] have revealed various shortcomings of traditional batch processing systems such as MapReduce [6] and Hive [7]. Batch processing systems suffer from latency problems due to the need to collect input data into batches before it can be processed. Under several application scenarios such as fraud detection in financial transactions and healthcare analytics involving digital sensors and Internet of Things (IoT), continuous data streams must be processed under very short delays [8]. This is because certain types of data streams such as stock values, credit card transactions, traffic conditions, time-sensitive patient data, and trending news rapidly depreciate in value if not processed instantly. Thus, the ability to handle and process continuous streams of data is becoming an essential part of building today's data-driven organizations.

Currently many tools exist for ingesting, processing, storing, indexing, and managing streaming data, which makes it a difficult task for practitioners to select the right combination of tools and platforms for building data stream analytics

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

applications [9]. The main motivation of this paper is to strive to alleviate this task by first identifying the major components of a modern Data Stream Processing System (DSPS). Before choosing a DSPS or any of its components, it is important to understand and evaluate why it is necessary, what is needed and what are the available options.

Data Stream Processing Engines (DSPEs) lie at the core of DSPSs and enable the definition and execution of stream processing pipelines. According to Russo *et al.* [10], DSPEs have emerged as key enablers to develop pervasive services that process data in a near real-time fashion. However, with the myriad of DSPEs that exist today, the large design space for DSPSs, and the intermingle of technical and marketing content from the different vendors and platforms, it is challenging to map design requirements to efficient and compatible hybrid solutions. This underscores the need for systematic studies to showcase advances in tools enabling data stream processing and open challenges, which form the basis of further research.

A number of literature reviews on DSPEs have emerged in recent years. Some of the early DSPE survey papers include the reports by Kamburugamuve *et al.* [4] and Bockermann [11], a selective survey of tools enabling data stream processing by Gorawski *et al.* [12] and a survey of open source near real-time technologies by Liu *et al.* [13]. Other surveys of DSPEs and their applications in various domains have been outlined in [14]–[16]. Recent surveys on DSPEs include a taxonomy of DSPEs by Zhao *et al.* [17], a global view of some popular DSPEs by Kolajo et al [18] and a performance comparison of some of the existing DSPEs with regards to time-series analysis by Gehring *et al.* [19]. Our work extends these recent studies by reviewing both design and implementation aspects of DSPEs unlike most of the previous surveys which focus either on the design, or the implementation aspect. Besides explaining the usefulness and the key features of DSPEs, this paper also provides insights on the design and implementation choices such as when DSPEs are not the most suitable paradigms, and when to choose distributed or single-node approach. The main contributions of this paper are:

1. An overview of DSPS including usability and architecture.
2. The listing of the key features of DSPEs based on an extensive study of the state-of-the-art DSPEs, which we used to propose a taxonomy of DSPEs.
3. A literature review of DSPEs based on the proposed taxonomy and a critical review of some of the popular DSPEs.
4. A comparison of DSPEs based on the key features.
5. A real-world DSPS use case scenario and the rational for selecting the most appropriate DSPE.
6. A discussion on the open research problems in this area.

The rest of the paper is organized as follows. Section 2 describes an overview of the architecture and the main components of a general DSPS. Section 3 presents the key features and taxonomy of DSPEs. A literature review of DSPEs encompassing the work of both academic researchers and industry professionals is presented in Section 4. A comparative analysis of some of the cutting edge open source distributed DSPEs (Storm, Spark Streaming, Flink, Kafka Streams, and IBM Streams) is also presented in Section 4. A use case scenario and design considerations for developing a DSPS is presented in Section 5 along with discussions and recommendations for choosing a DSPE for different applications. Section 6 summarizes the paper and draws the conclusion.

II. MOTIVATION AND AN OVERVIEW OF DSPS

Modern Data Stream Processing Systems (DSPS) try to combine batch and stream processing capabilities into a single or multiple parallel data processing pipelines [2]. The motivation behind developing such infrastructures is that massive processing can be done on historical data to train machine learning models using batch data analytic pipelines, which can then be deployed on real-time incoming data streams for scoring using a separate data analytic pipeline. Marz and Warren propose the lambda architecture [20], which includes batch and stream processing within a single framework. The lambda architecture was designed for applications that have delays in data collection and online processing due to the use of interactive interfaces or dashboards and data validation steps [21]. It is, however, limited in application because of the requirement of having to build, maintain, and integrate two separate systems. It is, therefore, desirable to reduce this complexity by providing a single interface at a higher level of abstraction to manage the underlying batch and streaming systems [2].

In this section, we first motivate the reader by describing data analytics scenarios that will necessitate the use of streaming analytics approach. Next, we give an overview of the general architecture of a DSPS, and finally, describe the main data processing components of a DSPS.

A. WHEN TO USE A DSPS

The traditional approach to processing data at scale is batch processing. Schreiner and Topolnik [22] described a typical example of a batch processing application in the financial sector where a bank collects and stores its transactional data in a data warehouse during the day. The bank then processes and analyzes the stored data offline after the closing time for decision making. Another scenario described by Dean in his book [23], uses a Web search system where crawlers constantly scrape the web and extracts knowledge but relies on batch processing to update its search index every hour. While both scenarios are simple and robust, the batch processing approach utilized in both cases clearly introduces a large latency between gathering the data and being ready to act upon it.

The goal of DSPS is to overcome this latency by processing big data volumes and provide useful insights into the data prior to saving it to long-term storage. It processes the live, raw data immediately as it arrives and meets the

challenges of incremental processing, scalability, and fault tolerance [22]. Streaming processing offers the competitive advantage of timeliness and should be used where real-time results matter or when the value of the information contained in the data stream decreases rapidly with time [23]. Typical example of scenarios that can benefit from DSPS include real-time analytics for systems monitoring and decision support. For example, hospital intensive care units have constant patient health monitoring systems, businesses implement financial news analytics for getting business insights fast, and fraud detection systems for ensuring secured transactions. Also, processing data as it arrives spreads out workloads more evenly over time, thereby, making the consumption of resources more consistent and predictable while enabling real time decision support [22].

Stream processing, however, raises new operational challenges and introduces new semantics for analytics [23]. To achieve low latency, a DSPS must be able to perform message processing without having a costly storage operation. It should also be able to handle imperfections in the data streams and process delayed, missing and out-of-order data. A detailed discussion on the requirement of a DSPS, is provided by Stonebraker *et al.* [24].

B. GENERAL ARCHITECTURE OF A DSPS

A DSPS can be deployed on a single machine consisting of multiple components, where each component directly calls the next component(s) in the processing pipeline. For greater throughput, efficiency, and fault tolerance, large stream processing systems are set up on multiple distributed machines with the help of additional software stack. A simple Extract, Transform, and Load (ETL) transactional DSPS architecture for processing IoT streams was proposed by Meehan *et al.* [25]. The architecture comprises three main components (i) stream collection for data ingestion, (ii) streaming ETL engine for real-time query processing, and (iii) an Online Analytical Processing (OLAP) backend for handling long-running queries. A data migrator component is used to connect the ETL and the OLAP components to facilitate data transformations.

The DSPS at Facebook [26] is relatively complex and powers many use cases such as the real-time reporting of aggregated voice of Facebook users, analytics for mobile applications, and insights for Facebook page administrators. It is made up of data sources such as mobile and web products; Scribe as a data distribution tool; stream processing systems such as Puma, Stylus, and Swift; and data stores such as Laser, Scuba, and Hive. The Facebook DSPS flow is a complex Directed Acyclic Graph (DAG) comprising of data streams from the mobile and web products that are fed into Scribe. Real-time data processing systems read from and write to Scribe. The data stores also use Scribe for data ingestion and serve different types of queries. According to Psaltis and de Assuncao *et al.* [8], [9], the architecture of a DSPS is generally multi-tiered and is composed of many loosely coupled components that include

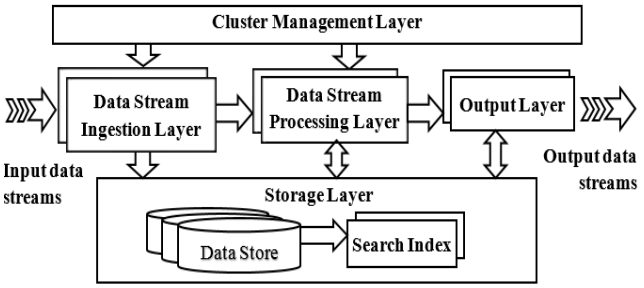

FIGURE 1. Architecture of a Data Stream Processing System (DSPS).

data sources, data collection systems, data storage systems, messaging systems, and stream processing and delivery systems.

C. COMPONENTS OF A DSPS

The input to a DSPS is a sequence of immutable and potentially infinite records which hold information about an event that happened. These records form what is called a data stream [22]. For most DSPSs, despite the diverse data domains and business logic, a generic data processing pipeline exists, which consists of (i) a *data stream ingestion layer* at the front-end, responsible for accepting streams of data into the DSPS, (ii) a *data stream processing layer,* which pre-processes and analyses data in one or more steps, (iii) a *storage layer,* which stores, indexes, and manages the data and the generated knowledge, (iv) a *resource management layer,* which manages and coordinates the functions of distributed compute and storage resources, and (v) an *output layer,* which directs the output data stream and knowledge to other systems or visualization tools. Fig. 1 represents a generic architectural blueprint of a DSPS. We also provide a concise description of each component.

1) DATA STREAM INGESTION LAYER

Data ingestion is the process of getting data streams from its source to its processing or storage system as efficiently and correctly as possible [25]. The ingestion layer ensures scalable, resilient, and fault-tolerant data distribution across the DSPS architecture from multiple input data streams, and decouples the input data sources from the other parts of the DSPS [9], [27].

There are many sources of input data streams [28]. These include data streams from various IoT devices such as sensors, video and other electronic monitors, social network Application Programming Interfaces (APIs), WebSockets, Representational State Transfer (REST) Web services, service usage logs, other stream processing systems, or any object which can collect and transmit time-sensitive data [8]. These input data streams can be comma or tab delimited text, graphs, JSON objects, WebSockets, events or any time-series data. The data stream ingestion layer deploys a variety of tools to accept different types of input data streams from one or more sources.

Queueing systems encompass the spectrum of messaging services, from the traditional message queuing products such

as MQTT, RabbitMQ, and ActiveMQ to the newer products such as NSQ and ZeroMQ [9], [29]. Apache Kafka and DistributedLog have grown to embody more than a message system, and both currently support publishing and subscribing to streams of records [8]. There are also many commercial stream ingestion systems including Scribe [26] developed at Facebook, Kinesis Data Firehose managed by Amazon Web Services (AWS), IBM WebSphere MQ and Microsoft Message Queuing [29].

2) DATA STREAM PROCESSING LAYER

The data stream processing layer is where the streaming data processing applications or jobs are executed. It can host loosely coupled disjoint applications or a DSPE or both. DSPEs generally offer a set of streaming data processing operators which can be configured and threaded together to build a stream data processing pipeline to analyze incoming data streams [30].

Many DSPEs exist today some of which are marketed by large software vendors while others were created as a part of open-source research projects. The main categories of DSPEs include Data Stream Management Engines (DSME), Complex Event Processing Engines (CEPE) and general-purpose DSPEs (GDSPE) [8].

Traditional Database Management Systems (DBMSs) are best equipped to run periodical queries over finite stored data sets. However, many modern data analytics applications such as financial data analytics, and network and sensor data monitoring, need to support continuous queries over dynamic unbounded data streams [31]. DSMEs provide extensions to DBMSs by enabling long-running queries over dynamic data, providing support for SQL-like operations and declarative interfaces. The basic requirements that a DSME should meet were outlined by Stonebraker *et al.* [24]. These requirements include the ability to (i) process continuous data on-the-fly without any requirement to store them, (ii) support high-level languages such as SQL, (iii) handle imperfections such as delayed, missing and out-of-order data, (iv) guarantee predictable and repeatable outcomes, (v) efficiently store, access, modify, and combine (with live streaming data) state information, (vi) ensure that the integrity of the data is maintained at all times and relevant applications are up and available despite failures, (vii) automatically and transparently distribute the data processing load across multiple processors and machines, and (viii) respond to high-volume data processing applications in real-time using a highly optimized execution path. Cutting edge DSMEs include SamzaSQL [32], KSQL, and SQLstream Blaze [30], [33].

CEPE is the core logic of Complex Event Processing (CEP) which denotes application of business rules to streaming event data (such as log or sensor data streams) associated with a timestamp [34]. This enables useful real-time actions to be taken over data streams [35]. A CEPE supports exploring relationships among events using specific rules as well as declarative interfaces [8]. Cutting edge CEPEs include Esper, StreamBase, and WSO2 CEP [30], [34], [36].

TABLE 1. Data Stream Processing Engines (DSPEs).

Type	NAME	License
DSME	SQLstream Blaze	Community and Commercial
	KSQL	Apache License 2.0
CEPE	StreamBase	Commercial
	Esper	GNU General Public License (GPL) (GPL v2)
DDSME	Spark Streaming	Apache License 2.0
	Flink	Apache License 2.0
Managed systems	IBM Streams	Commercial and Quick Start Edition

Generic-purpose DSPEs (GDSPE) are being developed to perform distributed stream processing with the aim of achieving scalable and fault-tolerant execution on cluster environments. Unlike DBMSs or CEPEs, GDSPEs do not provide declarative interfaces, thereby requiring developers to program applications rather than writing queries. Most modern GDSPEs support streaming SQL and CEPE either natively or through an add-on library. GDSPEs are commonly used to process high volume and velocity of streaming data.

A good DSPE should have the ability to process data in real-time and for any given time interval, as well as to peek into the data within a sliding window of time [37]. Cutting edge GDSPEs include Kafka Streams, Spark Streaming, Storm, Flink, and Samza as listed in TABLE 1 [8], [9]. There are also many commercial GDSPEs including IBM Streams, Amazon Kinesis, Azure Streams, Google Cloud Dataflow, Facebook's Stylus, Swift, and Puma [8], [26]. The detailed discussion of DSPEs will follow later in the paper.

3) STORAGE LAYER

DSPSs often store analyzed data, discovered patterns and extracted knowledge from different data processing stages for further processing [8]. Stored data must be organized and indexed along with external knowledge or metadata for executing subsequent analytics jobs.

Data storage solutions for supporting DSPS architecture ranges from traditional file systems such as HDFS and Baidu File System (BFS) to distributed file relational databases such as PostgreSQL, key-value stores such as Redis, in-memory databases such as VoltDB, document storage such as MongoDB, graph storage systems such as Neo4j, NoSQL databases such as Cassandra, and NewSQL such as CockroachDB [38].

4) RESOURCE MANAGEMENT LAYER

The resource management layer coordinates actions among compute and storage nodes and manages resource allocation and scheduling in distributed systems to enable parallel processing of high volume and velocity of data streams [39]. A typical data stream cluster consists of several compute nodes, and a cluster manager, which coordinates communication between the nodes [9]. Processes in

a distributed system can either exchange messages directly through a network or read/write into some shared storage. Coordination tasks such as electing a master node, managing group memberships, and managing metadata are important when building a multi-cluster distributed system [40]. Cutting-edge resource management tools include ZooKeeper [40], Yet Another Resource Negotiator (YARN), Mesos, Tupperware, and managed services such as Borg [39].

5) OUTPUT LAYER

The results from data stream processing pipelines can be directed to an application, another workflow, a visualization tool, or an alert or monitoring dashboard [8]. At the same time, data and extracted knowledge can be passed on to the temporary in-memory or permanent storage for subsequent analysis and query. Different data types and structures require specialized tool sets to visualize.

These tools were categorized by Liu *et al.* [41] into four groups: (i) *graph visualization* which includes static graph visualization (matrix representation and node-link diagrams) using tools such as TreeNetViz, and dynamic graph visualization (mental maps, animation techniques) using tools such as StoryFlow, (ii) *text visualization* which includes static text visualization (feature-based and topic-based) using tools such as Word Clouds or Phrase Nets, and dynamic visualization using tools such as SparkClouds, TextFlow and EventRiver, (iii) *map visualization* for geographic data exploration using tools such as BirdVis, and (iv) *multivariate data visualization* for generic data types using Multidimensional Scaling (MDS) projection tools. Visualization tools are often utilized in decision support systems (DSS) for both streaming and non-streaming data applications [27], [41], [42].

In this paper, we mainly focus on the comparative study of DSPEs which lie at the core of DSPSs and enable definition and execution of stream processing pipelines [10]. Key features of the distributed DSPEs are described in the next section.

III. DATA STREAM PROCESSING ENGINES (DSPEs)

DSPEs are complex stream processing engines which are composed of multiple components as described below in Section A. These components differ in the underlying programming models, configurations, and operations. Research in DSPEs has diverged [11] into (i) *query-based systems* such as NiagaraCQ [43], TelegraphCQ [44], and AsterixDB [45], (ii) *online distributed machine learning systems* such as Scalable Advanced Massive Online Analysis (SAMOA) [46], (iii) *streaming graph analytics systems* such as GraphJet [47], and (iv) *general purpose streaming data processing frameworks* [48] such as Flink and Spark Streaming, having low-latency [49] and a distributed parallel processing architecture [50], [51]. Here, we explain the key features of DSPEs in Section B and propose a taxonomy for comparing the state of the art DSPEs in Section C. For more information on use case scenarios and advantages of DSPEs such as Storm, S4, SQLstream, Splunk, Kafka, and SAP HANA refer to Khalifa *et al.* [49], Chen and Zhang [52]. Kejariwal *et al.* [53], and Sakr [54] provide detailed descriptions of design choices and challenges of selected DSPEs which include S4, Storm, Trident, M3, Samza, Akka, Flink, Spark Streaming, MillWheel, Pulsar, Heron, IBM Streams, Microsoft Trill, and Stream Insight.

A. COMPONENTS OF DSPES

DSPEs represent stream data processing pipelines as DAG of logically connected stream processing jobs, execute the DAG, and return the results. The fundamental components of DSPEs include (i) data sources and sinks, (ii) an application driver, (iii) a stream manager, and (iv) stream processors. Generally, *data sources* refer to the inputs to streaming engines while *sinks* refer to the outputs from the streaming engines. Some engines allow data ingestion from multiple sources whereas others support a single source only. The *application driver* represents the client code that communicates with the stream manager to submit program codes and data as jobs, controls the lifetimes of these jobs, and then collects the results. The *stream manager* is responsible for assigning the submitted streaming jobs to the available stream processors which execute the assigned jobs [9]. Each *processor* receives input from one or more data source queues, performs some computation on the input, and produces results that are added to the output queues [4].

Modern computing infrastructures rely on distributed systems for increased performance and reliability in managing huge volumes of data [55]. In a typical distributed system, it is common to have one or more worker nodes (stream processors), a cluster (stream) manager, and an application driver that appropriately allocates and controls task execution at the workers with the help of the manager.

B. KEY FEATURES OF DSPEs

The following sub-sections highlight the key features of DSPEs. These features include the programming model, data source interaction model, data partitioning strategy, state management, message processing guarantee, fault tolerance and recovery, deployment, and support base (e.g. community, high level language, advanced input sources, storage, and analytics).

1) PROGRAMMING MODELS

The programming model of a DSPEs consist of a processing task which receives stream items from a source, performs some processing, and then emits some items to a sink [22]. Specifically, a programming model is an abstraction of the DSPE that allows for stream transformations and processing. Given the unbounded nature of streaming data, it is therefore, not feasible to have a global view of the incoming streaming data [56]. Hence, data is processed in chunks by defining finite, bounded time windows on top of an infinite data stream.

A window in DSPEs is usually defined by either a time duration or a record count. The time-based window speci-

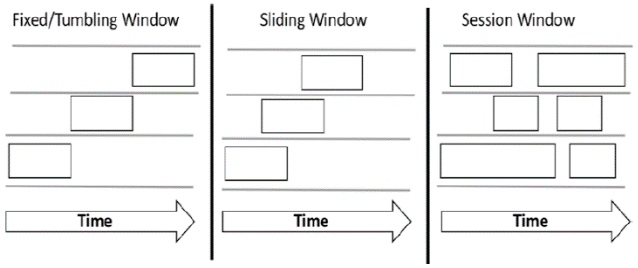

FIGURE 2. Windowing patterns [56].

fies a moving view that decomposes a stream into subsets and computes a result over each subset [57], [58] while the count-based window is sized by the number of events included in the window. This enables transformations to be executed on the records contained in the window [22].

Time-based windows are common in stream applications [59]. Luu [56] described three commonly used time-based windowing patterns that most modern DSPEs support as shown in Fig. 2. (i) Fixed or tumbling windows divide the incoming data stream into fixed-size segments, each with a window length, a start time, and an end time. Each incoming piece of data gets slotted into one and only one fixed window making it easy to perform aggregation transformations. (ii) Sliding windows divide the incoming stream of data into fixed-size segments, each with a window length and a sliding interval. If the sliding interval is the same size as the window length, then it is the same as the fixed window. If the sliding interval is smaller than the window length, then one or more pieces of data will be included in more than one sliding window; aggregation transformations in a sliding window will produce a smoother result than in the fixed window because of the overlapping of the windows. (iii) Session windows, unlike the fixed and sliding windows, do not have a predetermined window length. Instead, it is determined usually by a gap of inactivity that is greater than some threshold. A session is a burst of user activity followed by period of inactivity or timeout [22].

Many transformations on streams are carried out on each record independently and are often called stateless. Among the common types of stateless transformations are (i) Map, which transforms one record to another record, (ii) Filter, which filters out the records that do not satisfy a given predicate and (iii) FlatMap, which transforms each record to a stream and then flattens these streams into a single stream by combining the stream elements.

Transformations that require the processor to maintain internal state across the records and are called stateful. Common types of stateful transformations are (i) Aggregation, which combines all the records to produce a single value, (ii) Group-and-aggregate, which extracts a grouping key from the record and computes a separate aggregated value for each key, (iv) Join, which joins same-keyed records from several streams, and (v) Sort, which sorts the records observed in the stream. In stateful transformations, all the ingested records within a specified time window are included in the computation. The concept of windowing, meaningfully bounds the scope of the aggregation and can represent distinct or partially overlapped time frames [22].

In summary, DSPEs can do *real time streaming data processing* where tuples are processed as soon as they arrive or do *batch data processing* where time windows are used to split the tuples into batches or do both. A *batch* of stream elements for a given stream is a finite subset of the stream in a time interval. Distributed DSPEs usually process incoming data streams either natively (one tuple at a time) or in micro batches. A query or process is represented as a DAG consisting of a set of node operators, and the source and sink data stream operators. Operators can be either *stateless* or *stateful*. Most distributed DSPEs use the *continuous operator model* [60] where streaming computations are performed by a set of long-lived stateful operators. Each operator processes data stream records as they arrive by updating the internal state and sending new records in response [61].

2) DATA SOURCE INTERACTION MODEL

Understanding the interaction of data stream processing engines with data sources or message queuing systems is very important. The basic interaction model is the *push model* where a daemon process of a data stream engine keeps listening to an input channel. Another approach where the streaming engines continuously pull published data instances from stream sources such as WebSocket, Kafka, and Flume at a given frequency, is called a *pull model*. A challenge here is that the frequency of pulling and the speed of processing the data by the DSPEs should match the rate of data generation at the source to avoid data loss [9]. DSPEs can also interact with data sources using a combination of the two models.

3) DATA PARTITIONING STRATEGY

Large web companies run massive deployments of DSPEs in production. In this case, good utilization of resources is very critical. Skewed distribution of workloads within a DSPE can lead to poor resource utilization and inefficiency [62]. For efficiency, large volumes of data streams are partitioned and handled in a parallel or distributed environment. A partition is a logical chunk of data distributed across a cluster [63]. Partitioning strategies affect a system's scalability and data handling approach [4] and represents a distinctive feature of DSPEs as it varies for the different DSPEs. Data partitioning methods can be classified into horizontal and vertical approaches [64]. The horizontal method divides data into disjoint sets of rows. Horizontal partitioning method is further grouped into three techniques based on data set values, these are round-robin, range and hash partitions. According to Hamdi *et al.* [64], range partitioning is the most popular approach especially when there is a periodic loading of a new data. The vertical method divides data into vertical and disjoint sets of columns and can be categorized further into cost-based and procedural approaches. The cost-based method utilizes a cost model to predict the performance of the system and then selects the configuration to

optimize operational cost. The procedural approach applies scheduling and distributed computing procedures to define a good configuration.

4) STATE MANAGEMENT

A data streaming analytics program can be modelled as a directed tree where the analysis results are the roots, operators are the intermediate nodes, and data are the leaves. The data flows from the leaves through the operator nodes to the roots. Each operator node performs an operation that transforms inputs flowing through it into outputs. As described at the beginning of this section, operators can either be stateless or stateful. Stateless operators are purely functional, they produce output solely based on their input. On the other hand, stateful operators compute their output on a sequence of inputs and potentially use additional side information, maintained in an internal data structure called state. A state in a data stream application is a data structure that preserves the history of past operations and influences the processing logic for future computations [65].

In traditional DSPEs, state was separated from the application logic that computed and transformed the data, and the state information was stored in a centralized database management system to be shared among applications [66]. Handling state efficiently presents numerous technical challenges [65]. The state management facilities in various DSPEs naturally fall along a complexity continuum from naive in-memory-only choice to a persistent state that can be queried and replicated [9].

5) MESSAGE PROCESSING GUARANTEE

Message-processing guarantees is an important feature of DSPEs, and the semantics for this are *at-most-once*, *at-least-once*, or *exactly-once*. In an at-most-once semantics, a message is guaranteed to be delivered at most once, which implies that a message may get lost during routing and, if it is lost, no more attempts will be made to deliver it. This is the simplest guarantee semantics and it is the least fault tolerant. In an at-least-once semantics, greater fault tolerance is provided by ensuring that a message is delivered at least once until an acknowledgement of the delivery is received. Delivery may be attempted multiple times resulting in multiple instances of the same message being delivered in some cases at an additional cost for repeated processing. This is the most common guaranteed semantics offered by most data processing frameworks. In an exactly-once semantics, a message is guaranteed to be delivered exactly once with more acknowledgement checks along the way to prevent multiple delivery of the same message. This is the most desirable semantic. The choice of these semantics involves trade-offs between reliability needed for an application and the cost. For instance, for a streaming web analytics application a less complex faster process having weaker guarantees may be okay, while a streaming fraud detection application would possibly require a more reliable process with a stronger message guarantee [9].

6) FAULT TOLERANCE AND RECOVERY

Data stream processing engines must be operational for the length of time needed to recover from failures. Failures happen due to bugs in the system logic, problem in the network, crash of a compute node in a cluster environment including other hardware issues, reliance on a third-party software, and bottlenecks caused by the volume and speed of incoming data. The ability of a DSPE to keep running in the face of failures demonstrates its fault-tolerance capabilities. DSPEs should support good fault tolerance with minimum impact and overhead. Data loss (e.g. due to a crash in a node or for the data being in memory) and resource access loss (e.g. due to cluster manager fault) are common losses due to failures in DSPEs. Failure recovery incurs an additional resource demand on top of regular processing [61] and repeated processing causes overhead. For real-time applications, reprocessing from the start of the pipeline is impractical. Ideally a system should be able to restore itself to a previous state before the fault occurred and repeat only some of the failed components of the data processing pipeline. Fault tolerance in distributed DSPEs can be generally categorized into two types, *passive* (such as checkpoint, upstream buffer, and source replay) and *active* (such as replicas) approaches [67].

7) DEPLOYMENT

DSPEs can be deployed locally on a single machine, in a cluster, or in the cloud. Local deployment may lack the capacity to handle huge data volume and velocity. Cluster or cloud deployments of DSPEs are generally used for high volume and velocity data processing at the production-level. As for typical distributed applications, distributed DSPEs have one or more worker nodes where jobs are executed, a cluster manager which coordinates communications among the nodes [9], and a driver program that appropriately allocates and controls task execution at the worker nodes. ZooKeeper, Spark Master, Mesos Master, and YARN Resource Manager are some of the cluster management tools used by the different DSPEs.

8) COMMUNITY SUPPORT

Some of the DSPEs have vibrant community support groups and documentations, which greatly help in developing new data processing pipelines, codes to add to existing libraries, and configuring or customizing systems. The nature of developer and user base of a DSPE helps understanding its dynamics. Examples of community support include forums of code committers, contributors and Q/A, conference and summits for meetups and publications, and notifications on planned releases.

9) SUPPORT FOR HIGH LEVEL LANGUAGES

DSPEs with support for multiple high-level languages such as Java, Scala, Python, R, and SQL provide developers with greater choice of language to use for coding to reduce the implementation time of processing pipelines.

10) SUPPORT FOR ADVANCED INPUT SOURCES

DSPEs that support advanced input data sources such as local file systems, socket connections, databases (HDFS, S3, and Cassandra), and queuing tools (such as Kafka, Flume, Kestrel, NiFi, Cloud Dataflow, and RabbitMQ) can provide greater flexibility in linking with an existing big data ingestion pipeline to implement a DSPS. Refer to Section 2 for a detailed description of some of the advanced input systems.

11) SUPPORT FOR STORAGE SYSTEMS

DSPEs such as Spark Streaming provides a native in-memory storage, while others generally do not have their own data storage systems but provide data source and sink connectors to data ingestions systems such as Kinesis, Kafka, HDFS, and Cassandra, and search tools such as Solr and Elastic-Search. Refer to Section 2 for a detailed description of some of these storage systems.

12) SUPPORT FOR ANALYTICS

Besides the regular search and data storage systems, another desirable feature of DSPEs is support for analytical and continuous queries. The usual pipeline for mining and modelling data involves taking a sample or a snapshot from production data, cleaning and pre-processing the data, training and evaluating a model, and finally deploying the model to a production system. Most implementations of this traditional data mining pipeline such as WEKA and R are unable to cope with web-scale datasets. Hence the evolution of distributed data mining systems such as Mahout [30]. However, most data are now generated in the form of a stream and requires streaming models [31]. This involves analyzing data in near real-time or as soon as it arrives into the system. Most implementations of traditional stream processing systems such as Massive Online Analysis (MOA) are limited by the memory and bandwidth of a single machine [46]. Hence the evolution of distributed data stream mining systems such as the Scalable Advanced Massive Online Analysis (SAMOA), which enables the development of distributed streaming algorithms and executing them on multiple DSPEs [46].

C. TAXONOMY OF DSPEs

Although using DAGs for the abstract representation of streaming applications is followed by most streaming platforms, they differ in multiple ways. The field of data stream processing is very dynamic and new features of DSPEs get added to or updated in every new release. We propose a taxonomy as illustrated in Fig. 2 based on the features identified above and use the same to compare the state-of-the-art DSPEs in the next section.

IV. REVIEW AND COMPARISON OF DSPEs

Due to the rapidly evolving nature of streaming data analytic tools and big data processing systems in general, most recent studies about comparative performances of DSPEs are available in white papers published by industry researchers, blog posts of scientific analysts, and competitors' evaluations. We used scientific research publications from both the academia and the industry publications on a variety of online publication venues for the literature review.

A. LITERATURE REVIEW

Gualtieri and Curran [30] grouped about 26 criteria into three high-level categories namely current offering, strategy, and market offering to evaluate the strengths and weaknesses of 15 top commercial and open-source DSPE vendors and their products. The authors used a combination of product briefings and demos, vendor surveys, and customer reference calls for their study. This evaluation by Forrester uncovered a market with seven leaders (IBM, Software AG, SAP, TIBCO Software, Oracle, DataTorrent, and SQLstream) and eight strong performers (Impetus Technologies, SAS, Striim, Informatica, WSO2, Cisco Systems, data Artisans, and EsperTech).

The similarities, differences, implementation trade-offs, and the intended use-cases of five open-source distributed streaming frameworks were described by Zapletal in a theoretical overview [68] and an expert recommendation was made based on fault tolerance, state management and performance observations [69]. The study included Apache Storm, Apache Storm Trident, Apache Spark Streaming, Apache Samza, and Apache Fink. The author recommended that a careful evaluation of streaming application requirements should be conducted before choosing a framework.

Wähner [70] compared Spark Streaming, Flink, Beam, Storm, StreamBase, and IBM Streams. Matei *et al.* [71] compared Structured Streaming with Spark Streaming, Storm, Flink, Kafka Stream, and Google Dataflow. They concluded that Spark Structured Streaming facilitates integration into larger applications. A number of technologies have sprung up to solve the problem of real-time stream processing in a Hadoop environment [72]. Braida reported a comparative analysis of five top-level Apache projects, Apache Storm, Samza, Apex, Spark Streaming, and Flink, which provide stream processing capabilities, with IBM Streams. The study concluded that suitability of the options for stream processing depends on the specific use cases.

According to the study by Kamburugamuve *et al.* [4], the requirements of a good stream processing framework revolves around two important attributes; the latency of the system and the high availability of the system. The study also identified some expectations of a distributed DSPEs, which include (i) high data mobility, a key for maintaining low latency, (ii) high availability and data processing guarantees, (iii) effective data partitioning and parallel handling, (iv) support for high level languages and tools for querying data streams, (v) ability to recover from failures in the case of non-deterministic processing, (vi) support for data persistence, and (vii) ability to handle stream imperfections such as delay, out of order, duplicates or loss. These attributes and the architecture were used as the criteria to compare several streaming frameworks which included Apache Aurora, Borealis (no longer active as a research project), Storm, S4, and Samza.

The study by Gorawski *et al.* [12] developed three (StreamAPAS, THSPS, and AGKPStream) DSPEs and compared them with five (Borealis, Storm, Samza, StreamInsight, StreamGlobe) existing DSPEs based on the eight requirements of DSPE's identified by Stonebraker *et al.* 2005 as detailed in section 2. Gorawski *et al.* reported that all the tested DSPEs fulfilled requirements (i) and (viii). Requirements (iii), (iv), (vi) and (vii) were satisfied by most of DSPEs. Finally, requirements (ii) and (v) remained unfilled by most of the compared DSPEs. The technical report by Bockermann [11] compared several DSPEs (Storm, Samza, S4, MillWheel, Stratosphere, Streams) using the following criteria: message processing semantic, state handling and fault tolerance, scalability, support for distributed processing, and whether it is embeddable. Bockermann reported that the compared DSPEs differ from one another in terms of the guarantees they provide and the modelling capabilities they address. According to the study, Streams does not provide any fault-tolerance features by its own runtime, Storm, however, provides fault-tolerance similar to transactions and expects data to arrive in order. According to Bockermann, MillWheel builds upon low watermark timestamps and does treat all data streams as unordered.

Liu *et al.* [13] studied the shortcomings of Hadoop and the potential of lambda architecture with regards to data stream processing. Liu *et al.* also compared open-source messaging technologies and DSPEs including Hadoop Online, S4, Storm, Flume, Spark streaming, Kafka, Scribe, S4, HStreaming, and Impala based on their architectures, use case support, recoverability from failures, and license types. According to Liu *et al.*, despite their diversity, DSPEs share a great similarity especially in terms of using main memory and distributed computing technologies. Hesse and Lorenz [15] compared four DSPEs namely Storm, Flink, Spark Streaming, and Samza based on their language, stream abstraction, latency, throughput, message processing guarantees, and main components. According to Hesse and Lorenz, even though there are features that are common to all the compared DSPEs such as the use of Java Virtual Machine (JVM), a clear ranking cannot be created based on the presented results. Developing a use case and feature based ranking and recommendation system for DSPEs would be a good future research.

The study by Singh *et al.* [14] compared several stream processing solutions namely, Storm, Spark Streaming, S4, Amazon Kinesis, and IBM Streams based on the type of framework, implementation language, supported languages for application development, abstraction or primitiveness, data sources, model of computation or transformation, persistence entity, execution reliability, fault tolerance, latency, and vendor. Pääkkönen [3] compared AsterixDB and Spark Streaming with Cassandra in terms of their latency and throughput in processing semi-structured social media data streams. Pääkkönen reported that AsterixDB scaled better as more nodes were added to the cluster. AsterixDB with Java also achieved significantly higher throughput and lower latency when data feeds were utilized. AsterixDB stream processing was, however, delayed by batching of streamed tweets. The study by Yadranjiaghdam *et al.* [16] compared several DSPEs based on their application domains and recommend the extension of the use of DSPEs to other evolving application areas that may be beneficial to the society.

Chintapalli *et al.* [48] described the need for a benchmark for DSPEs and proposed one for selecting platforms for streaming data analytic needs. The benchmark focused on Apache Flink, Storm, and Spark Streaming within the context of a complete DSPS that utilized Kafka for ingestion and filtering of JSON events, and Redis for storing windowed count and timestamps of relevant events. Each of these frameworks has its advantages and disadvantages according to this study, for example, Storm and Flink have much lower latency than Spark Streaming at high throughput, while in terms of throughput rate, Spark Streaming can handle higher maximum throughput rate with its performance quite sensitive to the batch duration setting. Chintapalli *et al.* also added that tuning is required for Spark to achieve the desired latency to meet the service level agreements and recommended further work to look beyond just performance by incorporating other features such as security, and integration with other tools and libraries. Shukla *et al.* [73] proposed a benchmark suite for evaluating DSPEs for IoT applications and evaluated Storm using the same. According to Shukla *et al.*, the benchmark offers a set of realistic IoT tasks and applications that can be configured and utilized to evaluate DSPSs and DSPEs on public and private clouds. A recent study by Karimov *et al.* [74] proposed a framework to evaluate the performance of three DSPEs namely Storm, Spark Streaming, and Flink. Karimov *et al.* applied the proposed method to various use-cases and utilized the proposed benchmark for the evaluation of the DSPEs. Karimov *et al.* also proposed some recommendations for using DSPEs in various applications, for example, Spark Streaming is the recommended DSPE if a stream contains skewed data.

What is missing in all of the above studies is a systematic taxonomy based on which the DSPEs were compared. Zhao *et al.* [17] identified the need to develop a set of standard criteria for characterizing DSPEs. Zhao *et al.* developed a taxonomy for both open source and commercial data stream processing frameworks. The taxonomy developed by Zhao *et al.*, however, does not fully cover the technical depth and majority of the ingredients that are essential for characterizing data stream engines. In our proposed taxonomy in Section 3, we dived deeper into the key and technical features of the DSPE's such as programming and data models as well as the integration of these frameworks with other big data processing systems.

B. COMPARISON OF DSPEs

In this sub-section, we took a closer look at some representative leading-edge DSPEs selected based on their popularity [48], [74] and potentials. These DSPEs include Storm, Spark Streaming, Flink, Kafka Streams, and IBM Streams. We then compared these DSPEs based on a subset

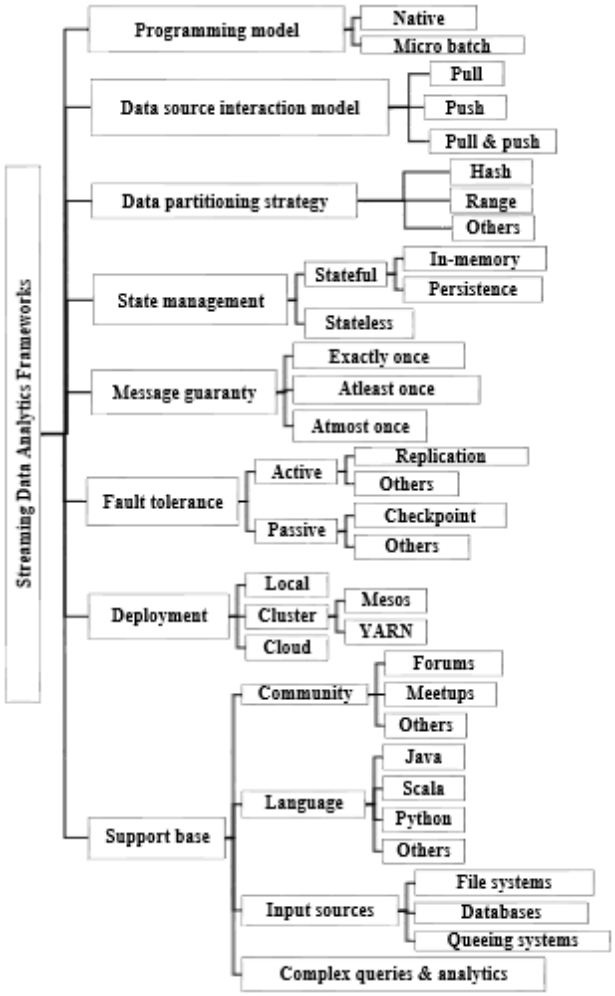

FIGURE 3. Taxonomy of streaming analytics frameworks.

of the key features in the taxonomy we have proposed and described in Section 3. The features that we found important for the comparison of the selected DSPEs are the programming model, data partition strategy, state management, message processing guarantees, fault tolerance and recovery. We also discuss additional criteria under the "Other" category.

1) PROGRAMMING MODEL COMPARISON

Storm is a native GDSPE. It has several processing layers such as Trident, Streams API, and SQL for processing data streams. There are three basic abstractions in Storm, these are spouts, bolts, and topologies. A spout is a source of streams in a computation, it reads data from message queues, databases, and other external sources and emit tuples to bolts in the topology without performing any processing. A bolt accepts a tuple from a spout, and then persist the data or performs some transformations (such as filtering, aggregation, or join operations) which may lead to the emission of new tuples to its output streams. A topology is a multi-stage stream computation network of spouts and bolts; each edge in the

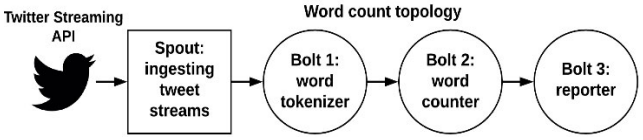

FIGURE 4. simple word count topology Storm.

FIGURE 5. Micro-batch model of Spark Streaming [75].

network represent a bolt subscribing to the output stream of some other spout or bolt. Fig. 4 is a simple Twitter stream wordcount topology in Storm. The topology consists of a single spout for ingesting tweet streams and three bolts for word tokenization, counting, and reporting.

Spark Streaming is a micro-batch GDSPE, an extension of the core Spark API. The main data abstraction in Spark is called the Resilient Distributed Dataset (RDD). Spark Streaming provides an abstraction which represents a continuous stream of data represented as a sequence of RDDs called *Discretized Stream* as shown in Fig. 5. Spark Streaming enables the processing of data streams as a series of small batch jobs thereby achieving end-to-end latencies as low as 100 milliseconds (ms).

However, from version 2.3, Spark introduced a Structured Streaming [76] library, built on the Spark SQL engine. Structured Streaming is based on a new low-latency processing mode called *Continuous Processing* which can achieve an end-to-end latency as low as 1ms. The evolution of Structured Streaming has made developers to consider Spark Streaming obsolete and not recommended for developing new streaming applications with Spark [77]. Therefore, further discussion in this paper is focused on Structured Streaming.

Flink is a native GDSPE with a support for batch data processing. The basic building blocks of Flink programs are streams and transformation operators mapped to streaming dataflows [78]. A stream is a continuous flow of data records while a transformation is an operation that takes streams as input and produce processed streams as output. Data streams in Flink can be created from message queues, socket streams, or files sources. Common examples of transformations on data streams include filtering, updating state, defining windows, and aggregating. Each dataflow starts with one or more sources and ends in one or more sinks which can be writing data to files or to other outputs. The dataflows in Flink as shown in Fig. 6, resemble arbitrary DAGs, with one or more sources and one or more sinks.

Programs in Flink are inherently parallel and distributed. Aggregating events (e.g., counts, sums on streams are scoped by windows, such as count over the last 50 seconds, or sum of the last 50 elements [78]. Windows can also be time, data driven, session driven as described in Section 3.

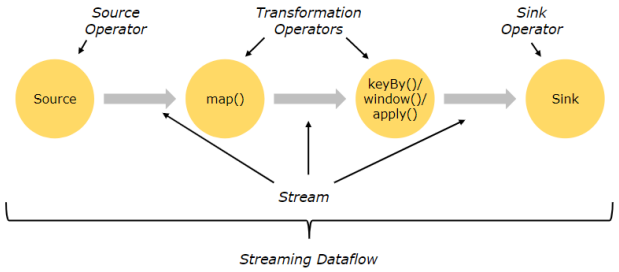

FIGURE 6. Flink's programming model [78].

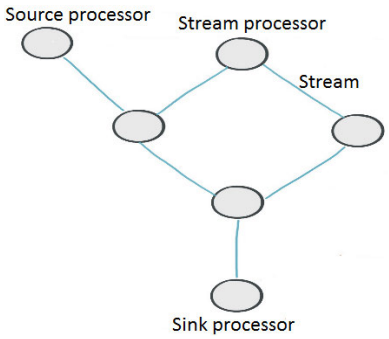

FIGURE 7. Kafka Streams topology [79].

Kafka Streams is part of the overall Kafka ecosystem which consists of immutable records or messages, stream of records called topics, consumers, producers, brokers, logs, partitions, and clusters. Kafka Streams is a GDSPE. A stream which represents an unbounded and immutable data records is the most important abstraction in Kafka Streams. A processor topology is the feature used in Kafka Streams to define the stream processing computational logic for an application. A topology is simply a graph of stream processors (nodes) that are connected by streams (edges) or shared state stores. There are two special processors (source and sink) in the topology as shown in Fig. 7. A source processor produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors. A sink processor then sends any received records from its up-stream processors to a specified Kafka topic. The processed results can either be streamed back into Kafka or written to an external system [79].

Kafka Streams offers two ways to define the stream processing topology. The first being the Kafka Streams Domain Specific Language (DSL) which provides the most common data transformation operations such as map, filter, join and aggregations out of the box. The second is the Processor API which enables the definition and connection of custom processors as well as interaction with state stores [79]. Windowing operations are available in the Kafka Streams DSL.

IBM Streams is a managed DSPE with a development environment, runtime and analytic toolkits. An IBM Streams application is a directed flow graph of operators [80]. As shown in Fig. 8, the fundamental building block of a Streams application is an operator.

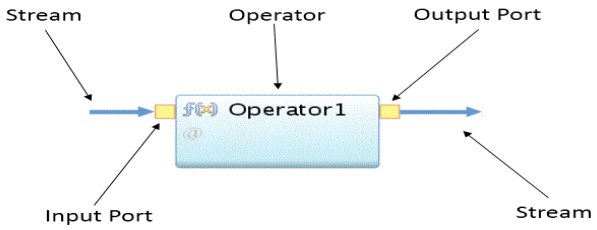

FIGURE 8. IBM Streams' programming model [80].

An operator can have one or more input and output ports. Individual records are structured list of attributes and their data types while a schema is a specification of data types and attributes in a tuple. The operator processes the records in memory and produces a new stream of records as output. The new stream of data is then emitted from the output port of the operator. The operator can process data on a tuple-by-tuple basis or on a window or batches of data.

2) DATA PARTITION COMPARISON

Storm provides complete control over how tuples are partitioned among the many tasks of a bolt subscribed to a stream using a feature called stream grouping. There are eight built-in and a custom stream grouping in Storm, these are (i) Shuffle, where tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples. (ii) Fields, where the stream is partitioned by the fields specified in the grouping. (iii) Partial key, where the stream is partitioned by the fields specified in the grouping but are load balanced between two down-stream bolts to provide better utilization of resources [62]. (iv) All, where the stream is replicated across all the bolt's tasks. (v) Global, where the entire stream goes to a single one of the bolt's tasks (usually the task with the lowest id). (vi) None, which does not care how the stream is grouped. (vii) Direct, which means that the producer of the tuple decides which task of the consumer will receive this tuple. (viii) Local, where tuples are shuffled to just those in-process tasks if the target bolt has one or more tasks in the same worker process, otherwise, it acts like a normal shuffle. (ix) Custom, which is achieved by implementing the Storm's custom stream grouping interface.

Spark automatically partitions RDDs and distributes the partitions across different nodes.*Hash* and *Range* are the common partitioning strategies supported in Spark. In hash partitioning, data is evenly spread across various partitions. However, in range partitioning, data tuples having key values within a given range will appear on the same machine [81].

Flink programs are inherently parallel and distributed. During execution, a stream in Flink has one or more stream partitions and each operator has one or more operator sub-tasks. Streams can transport data between two operators in a one-to-one pattern; this preserve the *same* partitioning and ordering of the elements. Streams can also transport data in a redistributing pattern by using partitioning strategies such as *hashing* or *random* re-partitions; this changes the original partitioning of streams [78].

Kafka Streams partitions data for processing while the messaging layer in the Kafka ecosystem partitions data for storing and transporting. In both cases, this partitioning enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of stream partitions and stream tasks as logical units of its parallelism model. Each stream partition is an ordered sequence of data records that maps to a Kafka topic partition. The Kafka producer picks which partition to send a record to based on the record's key. Records with the same key get sent to the *same* partition. The default partitioner for Java uses a *hash* of the record's key to choose the partition or uses a *round-robin* strategy if the record has no key [82].

IBM Streams supports a number of different partitioning methods using its DataStage technique [83]. These methods include (i) Round-robin, which assigns the first record to the first processing node; the second to the second processing node; and so on until it reaches the last processing node. It then starts over again. The round robin method always creates approximately equal-sized partitions; it is useful for resizing partitions of an input data set that are not equal in size. (ii) Random, a method that randomly distributes records across all processing nodes; similar to round robin, this method can rebalance the partitions of an input dataset to guarantee that each processing node receives an approximately equal-sized partition; however, the random partitioning has a slightly higher overhead than round robin because of the extra processing required to calculate a random value for each record. (iii) Same, a method which performs no data repartitioning and output from the preceding stage is fed as is to the next stage. (iv) Entire, where every instance of a DataStage receives the complete data set as input at every processing node. (v) Hash, a method which is based on a function of one or more columns (the hash partitioning keys) of each record; this method examines one or more fields of each input record (the hash key fields) and assigns records having the same values for all hash key fields to the same processing node. (vi) Modulus, which is based on a key column modulo the number of partitions; this method is similar to hash by field but involves simpler computation. (vii) Range, which divides a data set into approximately equal-sized partitions, each of which contains records with key columns within a specified range; this method is also useful for ensuring that related records are in the same partition. (viii) Db2, which partitions an input dataset in the same way that Db2 would partition it. (ix) Auto, which allows the DataStage to determine the best partitioning method to use; typically, DataStage would use 'round robin' when initially partitioning the data, and 'same' for the intermediate stages of a job.

3) STATE MANAGEMENT COMPARISON

Storm provides abstractions for bolts to manage (save and retrieve) the state of its operations. Currently, Storm has a default in-memory based state implementation and also a Redis backed implementation that provides state persistence. The only supported state implementation in Storm, at the time of writing this paper, is the KeyValueState, which provides key-value mapping.

At a high level, structured streaming tracks state in a manner similar in both its micro-batch and continuous modes. The state of an application is tracked using two external storage systems, a write-ahead log that supports durable, atomic writes at low latency, and a state store that can store larger amounts of data durably and allows parallel access (e.g., S3 or HDFS). Structured streaming uses these systems together to recover from failure [76].

Flink's ecosystem of modules and services built on its core offers different flavours of external state access and isolation. Each stream operation in Flink can declare its own state and update it continuously in order to maintain a summary of the data seen so far [66]. State information during Flink's operations is maintained in an embedded *key/value store* [78].

Kafka Streams provides applications with powerful, elastic, highly scalable, and fault-tolerant stateful processing capabilities. Kafka Streams provides state stores, which can be used by stream processing applications to store and query data, which is an important capability for implementing stateful operations. These state stores can either be a RocksDB database, an in-memory hashmap, or another convenient data structure [79].

IBM Streams operators can be configured to keep state information between subsequent tuples. This is achieved using tuple history, operator windows, operator custom logic, or primitive operators. The tuple history approach means that expressions in parameters or output assignments of operator invocations can refer directly to tuples received in the past. This is because history access maintains a list of tuples that can be subscripted by using the input port name. The operator windows approach uses a window which is a logical container for tuples recently received by an input port of an operator. The operator custom logic approach is made up of local operators' states that persist across operator firings and statements. An operator fires and statements get executed when an input port receives a tuple or a punctuation. Finally, the primitive operators which can either be generic or non-generic are user-written operators with some state mechanisms built into them. These are often controlled by the developers of the operators [84].

4) PROCESSING GUARANTEES COMPARISON

Storm offers several levels of message processing guarantees. These include best effort, at least once, and exactly once through Trident, a higher-level abstraction in Storm. Trident has consistent, exactly-once semantics and adds primitives for doing stateful, incremental processing on top of any database or persistence store.

Structured Streaming provides fast, scalable, fault-tolerant, end-to-end and exactly-once stream processing using a micro-batch processing engine. These guarantees can be achieved by selecting the relevant mode based on the application requirements without changes in the *Dataset* or *DataFrame* operations. An at-least-once guarantee

in Structured Streaming is achieved with the *Continuous Processing* model.

The mechanism for message processing guarantees in Flink ensures that even in the presence of failures, the program's state will eventually reflect every record from the data stream exactly once. Flink can guarantee exactly-once state updates to user-defined state only when the source participates in the checkpointing mechanism. In addition to exactly-once state semantics, Flink can also guarantee an end-to-end exactly-once record delivery, provided that the data sink participates in the checkpointing mechanism.

Starting from version 0.11.0, Kafka has added an end-to-end exactly-once processing semantics which guarantees that for any record read from the source Kafka topics, its processing results will be reflected exactly once in the output Kafka topic as well as in the state stores for stateful operations [79].

Consistent region is a recent concept introduced in IBM Streams for providing resiliency to streams applications by providing the ability to recover from failures. On a failure, an IBM Streams application is reset to its last successfully persisted state; this allows source operators in the application to replay tuples submitted after the restored state. Consistent region replay usually enables applications to achieve an exactly-once tuple processing semantic. For all other use cases, the replay enables applications to achieve at-least-once processing semantic [85].

5) FAULT TOLERANCE COMPARISON

Storm relies on a strategy called acking mechanism to replay tuples in case of failures. This mechanism tracks the completion of each tuple tree with a checksum hash. The checksum will be zero if all tuples have been successfully acknowledged. Every topology in Storm has a ''message timeout'' associated with it; hence, if Storm fails to detect that a spout tuple has been completed within that timeout, it will be considered to have failed the tuple and then it will be replayed later. The mechanism periodically (default to every second) checkpoints the state of the bolt to ensure that the bolt is initialized to its previous state during a system crash or restart.

Structured Streaming ensures an end-to-end exactly-once fault-tolerance guarantees through checkpointing and write-ahead logs. Structured Streaming checkpointing system uses a larger-scale state store to hold snapshots of operator states for long-running aggregation operators. The write-ahead log approach keeps track of which data has been processed and reliably written to the output sink from each input source. This log can be integrated with the sink to make updates to the sink atomic [76].

Flink implements fault tolerance using a combination of stream replay and checkpointing. A streaming dataflow in Flink can be resumed from a checkpoint while maintaining consistency or exactly-once processing semantics. This is achieved by restoring the state of the operators and replaying the events from the checkpoint [78].

Kafka Streams builds on a fault-tolerance capability integrated natively within the Kafka core. Kafka partitions are highly available and replicated, as such when a data stream is persisted to Kafka, it is available even if the application fails and needs to re-process it. Tasks in Kafka Streams leverage the fault-tolerance capability offered by the Kafka consumer to handle failures. If a task runs on a machine that fails, Kafka Streams automatically restarts the task in one of the remaining running instances of the application [79].

IBM Streams provides a number of facilities to allow recovery from failed hardware and software components [80]. IBM Streams provides support for the following failure recovery: (i) operator state by providing a checkpointing mechanism at regular intervals; (ii) failed processing elements by maintaining the status of each processing element using the Streams management services; (iii) failed application hosts by either restarting the host controller service or pausing/stopping the failed hosts; and (iv) management hosts by persisting the state of the critical management services and recovering it when the services are restarted. Streams applications can also achieve fault tolerance through user-applied consistent regions [86]. The summary of features of the selected five DSPEs considered in this study is described in TABLE 2.

6) OTHER COMPARISONS

Other features or criteria that are useful in deciding about what DSPE to use include the community base, third party integration, usability, and support for complex query processing and analytics. In terms of maturity and user base, Spark and Storm will have an edge over the other DSPEs; both are matured projects with bigger user base, more training materials, and more support for third-party libraries. The nature of the developer and user base of a DSPE helps in understanding its dynamics. For a project that requires complex stream processing and machine learning analytics, Spark and Flink will be the best options because of their built-in libraries for large-scale computations and their support for machine learning algorithms. For time critical applications that require a collection of services and hard real-time processing, Kafka Streams and IBM Streams are good options. We will describe a DSPS design use case scenario and our choice of DSPE for this use case based on this study.

V. DESIGN AND DEVELOPMENT OF A DSPS

A. USE CASE SCENARIO

Our industry-academic collaboration with Gnowit, a media analytics company is striving to develop a DSPS that can facilitate complex multilevel predictive analytics for real-time streaming text data from a variety of sources including the Web and social media data. Gnowit scrapes news articles from various sources for business intelligence [28]. The articles are collected on a continuous basis from several sources including news and blog websites, social media streams, and RSS/Atom news feeds, which need to be analysed to

TABLE 2. Comparison of DSPEs.

Criteria	Storm	Spark Streaming	Flink	Kafka Streams	IBM Streams
Version	2.0	2.4.3	1.8	2.3.0	4.3.0
Model	Native	Micro batch, continuous processing	Native	Native	Native
Partitioning	Shuffle, field, partial key, all, global, none, direct, local, custom	Hash, range	Same, hash, random	Same, hash, round-robin	Same, hash, random, round-robin, entire, modulus, range, Db2, auto
State management	Key value, redis	Write-ahead and other state stores	Key-value store	Key-value, hash map	Tuple history, operator windows, operator custom logic, primitive operators
Message delivery	Atleast-once, Exactly-once through Trident	Exactly- and At-least-once	Exactly-once	Exactly-once	Exactly-once
Fault tolerance	Checkpoint, Stream replay	Checkpoint	Stream replay, checkpoint	Stream replay	Checkpoint, consistent region, replay

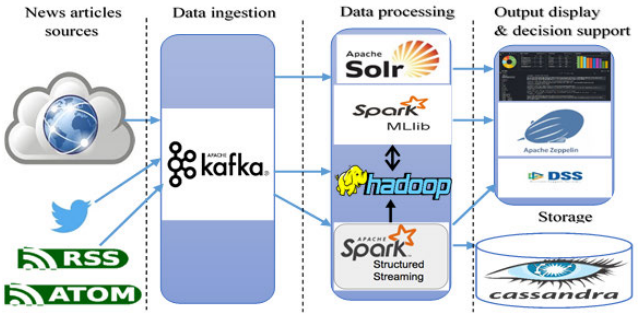

FIGURE 9. News articles stream processing use case.

determine correlations among news items, topics, objects, news publishers, and the people mentioned in the articles. The data processing for the DSPS would typically include Natural Language Processing (NLP), topic modelling from articles, sentiment analysis, and identification of duplicate and fake news.

B. DSPE SELECTION

With so many DSPE choices, Gnowit will have to factor in a number of things before picking the right DSPE for their stream processing needs. This is because the DSPE is the core component of a DSPS. Often, the choice of DSPE will determine the choice of other components of the DSPS. The main criteria that we considered for selecting the DSPE for this case study include support for a variety of programming languages, manageability and ease of integration with other data processing systems; other factors include the availability of a rich set of tools for data engineers like libraries for machine learning, complex event processing, graph processing and streaming SQL. Spark Structured Streaming and Flink turned out to be the two options that were well suited for this case study. Finally, we chose the Spark ecosystem because of its maturity and current user base.

Fig. 9 shows the different components that were selected to build such a DSPS. Data streams are ingested and pre-processed in Kafka [87], a distributed messaging and streaming platform. Kafka brokers publish the ingested streams as topics that are further processed and analysed using Solr, Hadoop, or Spark clusters [38]. Topics from Kafka are saved in a Hadoop Distributed File System (HDFS). Spark is used for distributed parallel execution of analytics processes to extract further knowledge from the data as needed. Solr is a real-time multilingual intelligent keyword searching and content clustering tool. Streaming data can be processed by Spark Structured Streaming and stored in efficient big data storage systems such as Cassandra. The extracted knowledge can be used to define machine learning models for predictive analytics to build a Decision Support System (DSS) sending outputs to a visualization tool. Thus, the framework supports processing and storage of large volumes of both streaming and batch data using Spark MLlib and Structured Streaming for decision support.

VI. CONCLUSION

A. SUMMARY

There are many existing and emerging applications that require real-time processing of high-volume heterogeneous data streams. There are also many open-source and proprietary systems for data stream processing. Extracting meaningful timely insights from unbounded data is very challenging. The large number of available systems is good but poses a major challenge in terms of selecting the right components or processing framework for different use cases. Understanding the required capabilities of streaming architectures is vital in making the right design or usage choice. This paper reports our contributions towards mitigating these challenges. We present a literature survey and a study of the components of data stream processing systems (DSPS). A critical review of the literature and key features of stream processing engines (DSPE) is presented which highlights some key differences in capabilities and applications of the DSPEs. We propose a taxonomy for categorizing DSPEs and use the same to compare cutting-edge open source and propriety DSPEs. Our ongoing work on streaming data analytics with our industry partner is presented as a use case that can serve as a guide for organizations and individuals planning to implement a real-time data stream processing and analytics framework. We discuss important gaps that were revealed in our study as future research.

B. OPEN CHALLENGES

The ever-increasing volume and highly irregular nature of data rates pose new challenges to DSPS. One such challenge is how to accurately ingest and integrate data streams from various sources and locations into an analytics platform. This demands new strategies and systems that can offer the desired degree of scalability and robustness in handling failures. Another interesting research gap revealed by this study is the lack of a well-organized benchmark for measuring the performance of DSPEs. There is a need for benchmarking DSPEs using attributes such as performance, security, and integration with big data tools and libraries for different use cases. Most of the previous surveys and comparative studies of DSPEs suggest that there are features that are common to most DSPEs such as the use of Java Virtual Machine (JVM), however, the ranking and recommendations for DSPEs for different use cases remain an active research area. There is also a need for big data stream mining systems that can combine machine learning algorithms and distributed stream processing in a single platform under an open source umbrella [35]. Spark and Flink are among the cutting edge DSPEs with these capabilities but more work is needed on the extension and integration of analytic libraries with DSPEs. For instance, there are many machine learning algorithms that are not yet supported in Spark Streaming and Structured Streaming.

The use case scenario discussed in this paper shows the inherent complexity of streaming data processing systems due to the need to include multiple data processing and management components, install these components on distributed computing resources that must be coordinated and managed, define and schedule data processing pipelines to be executed efficiently, and deliver the results in the desired format. There are many commercial or proprietary DSPSs which facilitate stream processing, but such systems provide limited flexibility for customization in choosing the different components of the DSPS. Researchers and developers can also build custom DSPS using open source software and tools, which can be extended as needed with new software code and deployed as a centralized or distributed multi-cluster system with preferred data storage and query systems.

The rapid growth of data in volume, veracity, and velocity has enabled exciting new opportunities and presented huge operational, data storage, and knowledge management challenges. There is a need for research on systematic comparative analysis and recommendations of databases or persistent stores for large streaming data. In addition, knowledge mining and management strategies are needed to identify, represent and store only the important data stream components and extracted knowledge instead of the massive raw data for further analytics and decision support.

REFERENCES

[1] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "SECRET: A model for analysis of the execution semantics of stream processing systems," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 232–243, 2010.

[2] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[3] P. Pääkkönen, "Feasibility analysis of AsterixDB and spark streaming with cassandra for stream-based processing," *J. Big Data*, vol. 3, no. 1, Apr. 2016, Art. no. 6. doi: 10.1186/s40537-016-0041-8.

[4] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu, "Survey of distributed stream processing for large stream sources," Grids Ucs Indiana Edu, School Inform. Comput., Indiana Univ., Bloomington, IN, USA, Tech. Rep., 2013.

[5] T. Dunning and E. Friedman, *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*. Sebastopol, CA, USA: O'Reilly Media, 2016.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[8] M. D. da Assunção, A. da S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. Comput. Appl.*, vol. 103, pp. 1–17, Feb. 2018.

[9] A. G. Psaltis, *Streaming Data: Understanding the Real-Time Pipeline*. Shelter Island, NY, USA: Manning, 2017.

[10] G. R. Russo, V. Cardellini, and F. L. Presti, "Reinforcement learning based policies for elastic stream processing on heterogeneous resources," in *Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2019, pp. 31–42.

[11] C. Bockermann, "A survey of the streamprocessing landscape," Lehrstuhl Fork Unstliche Intelligenz Tech. Univ. Dortmund, Dortmund, Germany, Tech. Rep. Version: 1.0, 2014.

[12] M. Gorawski, A. Gorawska, and K. Pasterak, "A survey of data stream processing tools," in *Information Sciences and Systems*. Cham, Switzerland: Springer, 2014, pp. 295–303.

[13] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," presented at the Proc. 18th Int. Database Eng. Appl. Symp., Porto, Portugal, 2014.

[14] M. P. Singh, M. A. Hoque, and S. Tarkoma, "A survey of systems for massive stream analytics," May 2016, *arXiv:1605.09021*. [Online]. Available: https://arxiv.org/abs/1605.09021

[15] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2015, pp. 797–802.

[16] B. Yadranjiaghdam, N. Pool, and N. Tabrizi, "A survey on real-time big data analytics: Applications and tools," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2016, pp. 404–409.

[17] X. Zhao, S. Garg, C. Queiroz, and R. Buyya, "A taxonomy and survey of stream processing systems," in *Software Architecture for Big Data and the Cloud*. Amsterdam, The Netherlands: Elsevier, 2017, pp. 183–206.

[18] T. Kolajo, O. Daramola, and A. Adebiyi, "Big data stream analysis: A systematic literature review," *J. Big Data*, vol. 6, no. 1, 2019, Art. no. 47.

[19] M. Gehring, M. Charfuelan, and V. Markl, "A comparison of distributed stream processing systems for time series analysis," in *Proc. BTW Workshopband*, 2019, pp. 205–214.

[20] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. New York, NY, USA: Manning, 2015.

[21] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct./Nov. 2015, pp. 2785–2792.

[22] V. Schreiner and M. Topolnik. Understanding Stream Processing. DZone Refcardz. Accessed: Aug. 28, 2019. [Online]. Available: https://dzone.com/refcardz/understanding-stream-processing

[23] W. Dean, *Fast Data Architectures for Streaming Applications*. Sebastopol, CA, USA: O'Reilly, 2016.

[24] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, 2005.

[25] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du, "Data ingestion for the connected world," in *Proc. CIDR*, Jan. 2017, pp. 1–11.

[26] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at Facebook," in *Proc. Int. Conf. Manage. Data*, Jun./Jul. 2016, pp. 1087–1098.

[27] M. O. Gökalp, K. Kayabay, M. Zaki, A. Koçyiğit, P. E. Eren, and A. Neely, "Big-data analytics architecture for businesses: A comprehensive review on new open-source big-data tools," Univ. Cambridge, Cambridge, U.K., Tech. Rep., 2017.

[28] S. Ge, H. Isah, F. H. Zulkernine, and S. Khan, "A scalable framework for multilevel streaming data analytics using deep learning," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Jul. 2019, pp. 189–194.

[29] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," in *Proc. 14th RoEduNet Int. Conf.-Netw. Educ. Res. (RoEduNet NER)*, Sep. 2015, pp. 132–137.

[30] M. Gualtieri and R. Curran, "The forrester wave: Big data streaming analytics, Q1 2016," Forrester, Cambridge, MA, USA, Tech. Rep. E-RES129023, 2016.

[31] 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," in *Data Stream Management*. Berlin, Germany: Springer, 2016, pp. 317–336.

[32] M. Pathirage, J. Hyde, Y. Pan, and B. Plale, "SamzaSQL: Scalable fast data management with streaming SQL," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 1627–1636.

[33] S. Geisler, "Data stream management systems," Schloss Dagstuhl-Leibniz-Zentrum Fuer Informatik, Wadern, Germany, Tech. Rep., 2013, vol. 5.

[34] L. J. Fülöp, G. Tóth, R. Rácz, J. Pánczél, T. Gergely, and A. Beszédes, "Survey on complex event processing and predictive analytics," in *Proc. 5th Balkan Conf. Inform.*, Jul. 2010, pp. 26–31.

[35] M. Dumoulin. *Better Complex Event Processing at Scale Using a Microservices-Based Streaming Architecture (Part 1) | MapR*. Accessed: Aug. 28, 2019. [Online]. Available: https://mapr.com/blog/better-complex-event-processing-scale-using-microservices-based-streaming-architecture-part-1

[36] S. Jayasekara, S. Kannangara, T. Dahanayakage, I. Ranawaka, S. Perera, and V. Nanayakkara, "Wihidum: Distributed complex event processing," *J. Parallel Distrib. Comput.*, vols. 79–80, pp. 42–51, May 2015.

[37] R. Thottuvaikkatumana, *Apache Spark 2 for Beginners*. Birmingham, U.K.: Packt, 2016.

[38] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surv.*, vol. 51, no. 2, p. 40, Apr. 2018.

[39] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," presented at the Proc. 10th Eur. Conf. Comput. Syst., Bordeaux, France, 2015.

[40] F. Junqueira and B. Reed, *ZooKeeper: Distributed Process Coordination*. Sebastopol, CA, USA: O'Reilly Media, 2013.

[41] S. Liu, W. Cui, Y. Wu, and M. Liu, "A survey on information visualization: Recent advances and challenges," *Vis. Comput.*, vol. 30, no. 12, pp. 1373–1393, 2014.

[42] R. Agrawal, A. Kadadi, X. Dai, and F. Andres, "Challenges and opportunities with big data visualization," in *Proc. 7th Int. Conf. Manage. Comput. Collective Intell. Digit. EcoSyst.*, Oct. 2015, pp. 169–173.

[43] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A scalable continuous query system for Internet databases," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 379–390, May 2000.

[44] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2003, p. 668.

[45] S. Alsubaiee *et al.*, "AsterixDB: A scalable, open source BDMS," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1905–1916, Oct. 2014.

[46] G. De Francisci Morales and A. Bifet, "SAMOA: Scalable advanced massive online analysis," *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 149–153, 2015.

[47] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "GraphJet: Real-time content recommendations at Twitter," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, Sep. 2016.

[48] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[49] S. Khalifa, Y. Elshater, K. Sundaravarathan, A. Bhat, P. Martin, F. Imam, D. Rope, M. Mcroberts, and C. Statchuk, "The six pillars for building big data analytics ecosystems," *ACM Comput. Surv.*, vol. 49, no. 2, Nov. 2016, Art. no. 33.

[50] D. Sun, G. Zhang, W. Zheng, and K. Li, "Key technologies for big data stream computing," in *Big Data: Algorithms, Analytics, and Applications*. Boca Raton, FL, USA: CRC Press, 2015.

[51] J. D. Bali, "Streaming graph analytics framework design," School Inf. Commun. Technol., KTH Roy. Inst. Technol., Stockholm, Sweden, Tech. Rep., 2015.

[52] C. L. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, Aug. 2014. doi: 10.1016/j.ins.2014.01.015.

[53] A. Kejariwal, S. Kulkarni, and K. Ramasamy, "Real time analytics: Algorithms and systems," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 2040–2041, Aug. 2015.

[54] S. Sakr, *Big Data 2.0 Processing Systems: A Survey*. Cham, Switzerland: Springer, 2016.

[55] M. Balduini, S. Pasupathipillai, and E. D. Valle, "Cost-aware streaming data analysis: Distributed vs single-thread," in *Proc. 12th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2018, pp. 160–170.

[56] H. Luu, *Beginning Apache Spark 2: With Resilient Distributed Datasets, Spark SQL, Structured Streaming and Spark Machine Learning Library*. New York, NY, USA: Apress, 2018.

[57] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2005, pp. 311–322.

[58] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *Proc. Int. Conf. Database Theory*. Berlin, Germany: Springer, 2005, pp. 37–52.

[59] U. Srivastava and J. Widom, "Flexible time management in data stream systems," presented at the Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst., Paris, France, 2004.

[60] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, Nov. 2013, pp. 423–438.

[61] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2013, pp. 725–736.

[62] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 137–148.

[63] M. A. Abbasi, *Learning Apache Spark 2.0*. Birmingham, U.K.: Packt, 2017, p. 356.

[64] S. Hamdi, E. Bouazizi, and S. Faiz, "Real-time data stream partitioning over a sliding window in real-time spatial big data," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.* Cham, Switzerland: Springer, 2018, pp. 75–88.

[65] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *Int. J. Very Large Data Bases*, vol. 27, no. 6, pp. 847–872, Dec. 2018.

[66] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache FlinkÂő: Consistent stateful distributed stream processing," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017.

[67] L. Su and Y. Zhou, "Progressive recovery of correlated failures in distributed stream processing engines," in *Proc. 20th Int. Conf. Extending Database Technol. (EDBT)*, 2017, pp. 518–521.

[68] P. Zapletal. *Comparison of Apache Stream Processing Frameworks: Part 1*. Accessed: Aug. 28, 2019. [Online]. Available: https://cloud.tencent.com/developer/article/1088152

[69] P. Zapletal. *Comparison of Apache Stream Processing Frameworks: Part 2*. Accessed: Aug. 28, 2019. [Online]. Available: https://cloud.tencent.com/developer/article/1088157

[70] K. Wähner. *Streaming Analytics Comparison of Open Source Frameworks, Products, Cloud Services*. Accessed: Aug. 28, 2019. [Online]. Available: https://www.slideshare.net/KaiWaehner/streaming-analytics-comparison-of-open-source-frameworks-products-cloud-services

[71] M. Zaharia, T. Das, M. Armbrust, and R. Xin. *Structured Streaming in Apache Spark*. Accessed: Aug. 28, 2019. [Online]. Available: https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

[72] A. Braida. *Stream processing and the IBM Open Platform*. Accessed: Aug. 28, 2019. [Online]. Available: http://www.ibmbigdatahub.com/blog/stream-processing-and-ibm-open-platform

[73] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 21, 2017, Art. no. e4257.
[74] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," Feb. 2018, *arXiv:1802.08496*. [Online]. Available: https://arxiv.org/abs/1802.08496
[75] Apache Spark. *Spark Streaming Programming Guide*. Accessed: Aug. 28, 2019. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html
[76] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in Apache spark," in *Proc. Int. Conf. Manage. Data*, Jun. 2018, pp. 601–613.
[77] J. L. Book, *The Internals of Spark Structured Streaming*. Oro Valley, AZ, USA: Gitbook, 2019. [Online]. Available: https://jaceklaskowski.gitbooks.io/spark-structured-streaming
[78] *Dataflow Programming Model*. Accessed: Aug. 28, 2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/concepts/programming-model.html
[79] *Kafka Streams*. Accessed: Aug. 28, 2019. [Online]. Available: https://docs.confluent.io/current/streams/index.html
[80] C. Ballard, K. Foster, A. Frenkiel, B. Gedik, M. P. Koranda, S. Nathan, D. Rajan, R. Rea, M. Spicer, B. Williams, and V. N. Zoubov, "IBM infosphere streams," in *Harnessing Data in Motion*, 1st ed. New York, NY, USA: International Technical Support Organisation, 2010.
[81] L. P. Gangadharaiah. *An Intro to Apache Spark Partitioning: What You Need to Know*. Accessed: Aug. 28, 2019. [Online]. Available: https://www.talend.com/blog/2018/03/05/intro-apache-spark-partitioning-need-know
[82] J.-P. Azar. *Kafka Producer Architecture—Picking the Partition of Records*. Accessed: Aug. 28, 2019. [Online]. Available: https://dzone.com/articles/kafka-producer-architecture-picking-the-partition
[83] *IBM Streams Quick Start Edition v4.3*. Accessed: Aug. 28, 2019. [Online]. Available: http://ibmstreams.github.io/streamsx.documentation/docs/4.3/
[84] C. Ballard, K. Foster, A. Frenkiel, B. Gedik, M. P. Koranda, S. Nathan, D. Rajan, R. Rea, M. Spicer, B. Williams, and V. N. Zoubov, *IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution*. Austin, TX, USA: IBM Systems Worldwide Client Experience Center, 2012.
[85] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyuce, "Consistent regions: Guaranteed tuple processing in IBM streams," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1341–1352, 2016.
[86] S. Schneider, B. Gedik, and M. Hirzel, "Language runtime and optimizations in IBM streams," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 38, no. 4, pp. 61–72, 2015.
[87] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

HARUNA ISAH (M'18) received the B.Eng. degree in electrical and electronics engineering from the University of Maiduguri, in 2008, and the M.Sc. degree in software engineering and the Ph.D. degree in computing from the University of Bradford, U.K., in 2012 and 2017, respectively. He is currently an IBM-SOSCIP Postdoctoral Fellow with the Big-Data Analytics and Management Laboratory (BAM Lab), School of Computing, Queen's University, Kingston, ON, Canada. His current research interests include big data and streaming analytics, machine learning, graph analytics, and natural language processing. He was a recipient of the National Information Technology Development Fund and the Commonwealth Scholarships.

TARIQ ABUGHOFA received the B.Sc. degree in software engineering and information systems from Damascus University, in 2013, and the M.Sc. degree from the School of Computing, Queen's University, Kingston, ON, Canada, with the thesis title of Community Detection for Large-Scale Dynamic Graphs. He was a Backend and Full-stack Software Engineer with Blue Kangaroo.

SAZIA MAHFUZ received the B.Sc. degree in computer science from the University of Dhaka and the M.Sc. degree in computer science from Acadia University, in 2017. She is currently pursuing the Ph.D. degree with the School of Computing, Queen's University, Kingston, ON, Canada, with the Ph.D. title of a hybrid autonomous data analytics framework using a semantic data profiling strategy.

DHARMITHA AJERLA received the B.Tech. degree in computer science and engineering from the VNR Vignanajyothi Institute of Engineering and Technology, in 2017, and the M.Sc. degree from the School of Computing, Queen's University, Kingston, ON, Canada, with the project title of Edge computing for wearable sensor data analytics.

FARHANA ZULKERNINE received the Ph.D. degree from the School of Computing, Queen's University. She is currently an Assistant Professor and the Coordinator of the Cognitive Science Program with the School of Computing, Queen's University. She is a member of the Professional Engineers of Ontario. She has more than 15 years of international work experience in three continents in software design, analysis, and research. Her current research interests include big data analytics and management, cognitive and cloud computing.

SHAHZAD KHAN received the B.Sc. degree in computer science from the Lahore University of Management Science (LUMS), Lahore, Pakistan, the M.Sc. degree in information studies from Syracuse University, Syracuse, NY, USA, and the Ph.D. degree in computer science from the University of Cambridge, U.K. He currently leads the artificial intelligence, machine learning, and natural language processing practices with Gnowit. He is the author of more than 22 peer-reviewed academic publications. He holds five patents. He is also a Mentor of the Ottawa chapter of the Silicon Valley founder's Institute and on the board of advisors for four AI-focused startups in Canada and the US.