Distributed Stream Processing

(Real-Time Stream Processing For Big Data)

Dirk Löwenstrom
dirk.loewenstrom@haw-hamburg.de
Hamburg University of Applied Sciences, Department of Computer Science
Berliner Tor 7
20099 Hamburg, Germany

***Abstract*—Stream processing is used to handle large sets of unbounded data, typically referred to as *events*, that arrive with high velocity in a mostly semi- or unstructured form. To obtain reliable information from these events, various window operations are used to group related events over a period of time, even if the events do not reach the processing pipeline in the order they occurred. To build a high-performance but reliable Data Stream Processing System (DSPS) tools like Apache Kafka are used to enhance the overall capabilities of the stream processors. In this paper, we will briefly glance over the basics of (distributed) stream processing, the architectures in which they are used, as well as open questions and interesting research topics.**

***Keywords*—distributed stream processing, Big Data, streaming machine learning, benchmarking**

I. Introduction

Stream processing engines (SPE) have become an important part of enterprise Big Data architectures by extracting business value from data in motion, reducing the processing time of an individual item and offer near real-time decision making. Data from the ever-growing list of IoT sensors, smartphones, wearables and other connected devices emphasize the *velocity* part of Big Data handling which led to a shift from traditional batch oriented processing such as MapReduce [28] on Hadoop [53] towards a more ad-hoc approach because the amount of data and the speed at which it is generated rules out other approaches. In addition, the value of data provided from such sources decays quickly, making high latency in those problem domains unacceptable. Despite the urge for real-time processing, SPE must be able to cope with the amount of rapidly generated data. Therefore, high throughput is also an essential property of a SPE in order to keep up with incoming data streams. For example Twitter [14] needs to handle more than 2.8 billion write requests per minute, storing 4.5 petabytes of time series data. LinkedIn [64] generating over 1 trillion messages per day using Kafka [40] and recently Steven Wu mentioned the use of the Apache Flink [4] system for the data platform at Netflix which handles over 3 trillion events per day. Other common use cases for stream processing include clickstream analysis or enrichment, ETL workloads and predictive analytics (fraud, pattern recognition) using incremental updated prediction models [20].

However the interest in processing data in motion is not new. Many advances (algorithm improvements, querying on streams or machine learning) in event processing (EP) and complex event processing (CEP) were made a decade ago [15, 19, 30] and SPE has gone through a long evolution. From an extension of traditional DBMS as main-memory database systems or rule engines triggering on new events, over using SQL on streams by exploiting the relation between streams and relations (TelegraphCQ [23], STREAM [15]), to engines like Aurora [19] that use a directed acyclic operator graph (DAG) based approach for processing streaming data on a single or multi-node system with fault tolerance and distribution capabilities (Borealis [10]). Although those systems can provide continuous query capabilities, high-level SQL-like interfaces and latency in the order of milliseconds their deployment usually only spawns a few nodes and lack support to maintain a high-scalable system needed to solve aforementioned requirements for throughput and latency [63]. This has led to the development of a number of recent systems with their own trade-offs that cover the requirements [56] of large-scale distributed stream processing engines on big cluster environments. A not exhaustive list includes: MillWheel [12], Kafka Streams [6], Cloud DataFlow [13], Apache Apex [1], Apache Spark Streaming [3], Apache Flink [4], Apache Storm/Trident [8], Apache Heron [45] and Apache Samza [7]. Several SPE also support both batch and stream processing capabilities, leading to a simpler architectures than the widespread Lambda Architecture [47] using a single SPE for data processing. Recent work on the Apache BEAM [2] project goes even further by providing a unified programming API based on the work of Akidau et al. and the dataflow model. In BEAM batch or stream workloads can be described using an abstraction of stream with different runners, while the ”real” implementation using one of the previous listed SPEs. Notwithstanding using BEAM or the underlying SPE directly, these systems still remain challenging to use in practice and place significant demands on things like debugging, maintenance, elasticity, scalability as well as performance evaluation [17, 18, 27, 33]. The rest of this paper is organized as follows.

First we will glance over the basics of stream processing engines and the critical characteristics needed to use them in large-scale systems. The main part will look at the different SPEs, describing their architectural and execution model as well as comparing the trade-offs they make. Finally we briefly summarize open issues and interesting topics in the area of distributed stream processing.

II. Basics & challenges

Before we dive into the details of some of the modern SPEs, we take a look at architectures that are used in Big Data systems. A general data processing pipeline consists of the following parts: data sources emit data to the system where they are stored in a distributed, fault-tolerant file system like HDFS [53] or queueing system like Kafka [6] followed by arbitrarily processing, serving and some form of visualization. In this paper we focus on the processing step, especially what options exists to realize stream processing. To better understand the trade-offs, we first need to distinguish batch and stream processing.

Batch vs. Stream Processing

Batch processing handles the *volume* part of Big Data by periodically execute batch jobs on large potion of the incoming data in a cost-effective and efficient way [31]. Because batch jobs operate on the complete data, optimized parallel algorithms like MapReduce [28] can be used to get correct results at the cost of overall higher latency due to the delayed execution. In contrast, streaming systems tackle *velocity* and provide a low end-to-end latency by processing events at-motion. However the benefits introduce a series of challenges [56] that need to be solved when reasoning about distributed stream processing systems (DSPS) and unbounded streams of events:

- **scaling:** the system must be able to scale-out from a single-node to a multi-node cluster and provide a way to deploy processing logic to that cluster.
- **availability:** in contrast to batch processing, streaming uses long-running jobs to process items immediately, making downtime unacceptable.
- **stragglers:** since the system can process data only as fast as its slowest component, it need a way to handle *stragglers* by rebalancing or replacing misbehaving components.
- **backpressure:** is important because it defines how the system responds to overload and maintain operational.
- **fault-tolerance:** the DSPS must be fault tolerant and automatically return to the normal state in the event of a fault.
- **state management:** for stateful stream processing the system need to maintain state of calculated results and provide a way to recover the intermediate results on cluster failure.
- **out-of-order processing:** the DSPS need techniques to reason about the skew between *event time* (the time where the event actually occurred) and *processing time* (the time at which the event is observed by the system) [60]. Network latency or hardware errors can cause events to arrive late and out-of-order. Therefore a way to re-order late events and process them in the right time window is essential for workloads that require strict ordering to output correct results. In addition, the system only operates on partial data and cannot make assumptions on the completeness of events in a certain time period. Therefore, it must be able to work on incomplete data, leading to approximate results compared to batch processing. Handling late events is always a trade-off between result latency and result accuracy [63].
- **processing guarantees:** as part of fault-tolerance a DSPS defines delivery guarantees for the processed items: *at-least-once*, *at-most-once* and *exactly-once* [60]. At-most-once provides the lowest guarantee since messages are delivered zero or at most once. No attempts are made to retry or retransmit events which may cause messages to be lost. At-least-once guarantees that a message will be processed even in the presents of failures by using a replayable source like Kafka to retry processing. As the name implies the SPE may process an event twice, thus allowing duplicates to occur. In exactly-once processing messages can neither be lost nor duplicated. Famous approaches include at-least-once plus dedupluciation and distributed snapshots that periodically checkpointing processing state to a durable backend. The term exactly-once however is misleading for what it ensures, because it cannot guarantee that arbitrarily user-defined logic executed for an event happens only ”exactly-once”, instead it ensures that the state of the data processing pipeline is only affected once by an event and that this state is recovered correctly [51]. Most systems that offer exactly-once also rely on a replayable data source to replay events past the latest checkpoint.
- **latency:** achieved by SPE only display a portion of the overall end-to-end latency for Big Data processing systems. When the event is generated at the source it need to be sent to the ingestion system which forwards the event stream to the streaming processing layer. After processing, results are passed to downstream systems for storage and serving. All these steps contribute to the overall latency of the pipeline.

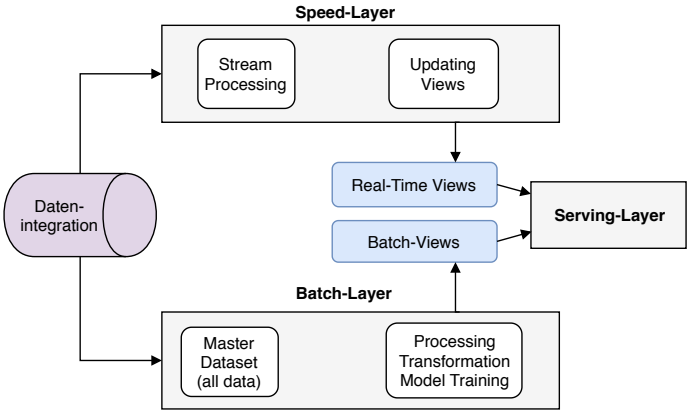

Fig. 1. Lambda Architecture [47]

Lambda & Kappa Architecture

As an attempt to merge the benefits of both worlds, Marz proposed the Lambda Architecture that combines batch and stream processing in one architecture. Distilled to its essence, the Lambda Architecture complements the slow, but correct results form batch processing with a real-time layer for fast, but not necessary correct results. As shown in Figure 1 incoming data flows two paths. The *batch layer's* job is to persist data to a fault-tolerant, highly scalable storage system like HDFS [53] and periodically execute batch jobs on large datasets, while the *speed layer* operates on recent data, providing low-latency results. Finally the *serving layer* merges the output from both layers to build a complete result. However using this kind of architecture essentially means maintaining two systems, their individual computation logic and the not trivial effort to merge the results. Although hybrid tools that support batch and stream processing exists [3, 4, 50], most systems rely on Hadoop and need to employ a separate streaming system [63]. Duplicated logic can be minimized using higher abstractions like Summingbird [21], but this still requires managing two separate systems. In contrast, the Kappa Architecture [39] makes no distinction between batch and stream processing by treating everything as a stream (batch is just a special case of stream over a fix window of historical data). Mainly all data is stored in a highly scalable messaging systems like Kafka [6] which provides an ordered, fault-tolerant stream of events that can be replayed into the system as needed. The core of the data processing pipeline is a SPE that must be able to cope with incoming data at high rate and achieving high throughput. Instead of having two data paths, all data processing takes place in the SPE, with the results being recalculated only for computation logic updates by replaying historical data using the aforementioned messaging system [39]. Also the Kappa Architecture come with disadvantages. The effort to replay the entire history is proportional to the stored data volume, placing a lot of complexity on the ingestion system to retain large datasets. Though the Kappa Architecture has a limited use case for application that do not require unbounded retention times of data [63].

III. Distributed Stream Processing systems

The following sections will focus on the current state of distributed stream processing engines, concentrating on their architectures, state handling and processing guarantees.

A. Apache Storm

The first SPE we analyze is Apache Storm [8] that provides an abstraction for stream processing, like MapReduce [28] did for batch processing. Storm [59] defines a data processing pipeline as an indefinitely running *topology*, a directed computation graph that defines how incoming data (unbounded sequence of tuples) flow through the system and what processing should be applied to it. The topology itself consist of *spouts* and *bolts* as seen in Figure 2. Spouts handle the ingestion part by reading data from external sources or pull tuples from message queues like Kafka [6] and emit them

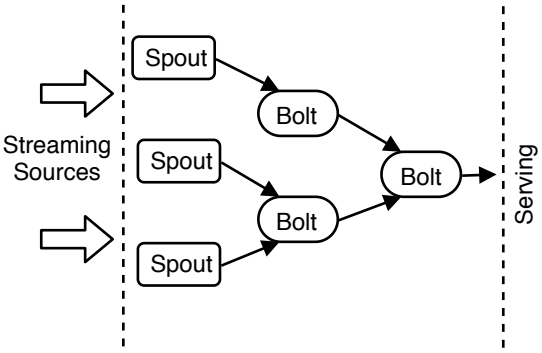

Fig. 2. Storm topology [59].

to the downstream bolts which execute the computation logic (e.g. filtering or user-defined functions (UDFs)), write results to external storage or pass data subsequent bolts.

For clustering storm uses a master-slave architecture, composed of Zookeeper [9] a distributed key-value store for reliable coordination and cluster state, a master node, called Nimbus that is responsible to observe cluster members, distribute tasks among slave nodes and take action in case of failing nodes by rescheduling tasks or restarting worker processes [18, 59, 63]. Each Storm slave node has a supervisor to orchestrate workers (JVM processes) which execute the actual tasks (the implementation of a spout or bolt described in the topology).

State Management & Fault-Tolerance

To prevent data loss Storm [8] is able to back up the progress of tuples by persisting state in-memory or synchronously forwarding the current state to an external key-value store like Redis. While this ensures that no intermediate result is lost, it will introduce very high-latency penalties as the size of the state increases, making it only feasible for small state to preserve the low-latency guarantees [31]. In case of a failing worker, the Nimbus simply restart it or if the whole slave is misbehaving it reschedules tasks as necessary. To cope with system overload, Storm can generate backpressure when the input buffer of a bolt exceed a defined watermark, throttling the previous steps until the system recovers [8].

Guarantees

Storm offers neither *exactly-once* semantics nor guarantees the order in which tuples are processed, but is able to achieve *at-least-once* by tracking the lineage of each tuple and acknowledging in each spout and bolt that the tuple was processed. This doubles the message overhead causing an enormous impact on performance [26].

Being a native stream processor Storm generally provides low-latency results at the cost of losing certain processing guarantees. In production use cases however the minimal achievable end-to-end latency is usually above 50ms due to network latency and garbage collection [32]. In addition, Storm is only able to express stream operations via a limited, low-level API which resulted in the Trident extension.

B. Storm Trident

Trident introduces an abstraction layer on storm that provides a higher-level API for processing stateful streams (e.g. joins, aggregates, grouping) and through the use of *micro-batching* a way to implement exactly-once processing as well as ordering guarantees between each batch. In contrast to Storm, Trident topologies are DAGs and tuples are processed in batches, not one-at-a-time, thus increasing throughput at the cost of increased latency. Each batch is the result of splitting the dataset into partitions, subdividing each partition into batches that are processed by one node in the cluster [31].

Guarantees

To realize exactly-once processing Trident uses the acknowledgment feature of Storm and guarantees that each tuple exists only once by storing additional information alongside the maintained state and updating it transactionally [63]. In order to maintain these guarantees, state updates must be performed in strict order, making this mechanism similar to Storm only feasible for small state.

C. Heron

Heron [41] is the direct successor of Storm at Twitter to address critical challenges related to scalability, isolation and debugging while maintaining compatibility with the Storm API. One of the main problems associated with Storm is the fact that workers from different topologies could interfere each other on the same machine, leading to a separate infrastructure for topologies in order to achieve isolation at the expense of inefficient use of resources. [45]. To solve this problem, Heron provides a cleaner mapping of logical units to physical execution processes instead of bundling multiple components of a topology into a single operating system process. In contrast to Storm, Heron topologies are process-based running each process in isolation and it is possible to specify fine-grained resource allocation for individual components to avoid over-provisioning the cluster as well as running a topology alongside other critical services [41]. Heron follows the same data model as Storm describing stateful topologies of spouts and bolts, optionally acknowledging the processing state of tuples for fault-tolerance and at-least-once guarantees [58]. Much like Storm, using this mechanism increases the overall latency, while disabling it only offers at-most-once guarantees.

Figure 3 shows the overall architecture of Heron, which in its basis consists of a Topology Master, Containers, Stream Manager, Heron Instance, Metrics Manager, Zookeeper [9] and one of the supported schedulers [5] that spawns the topology. When submitting a topology the scheduler allocate required resources, spawns several worker containers and a dedicated container marked as Topology Master that acts as the master node for the topology. The workers contain a Stream Manager for internal and external routing of tuples to or between Heron Instances and a Metric Manager exporting metrics to a central monitoring system. Stream Managers notify their Topology Master about scheduled containers that distribute the combined

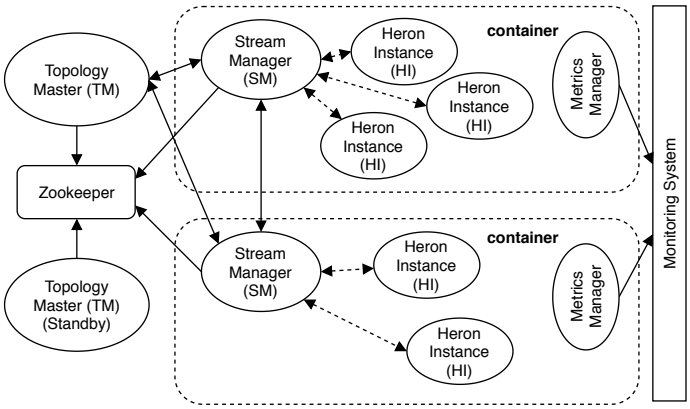

Fig. 3. Heron topology architecture [41].

physical execution plan to all known Stream Managers which form a fully connected graph [45]. Individual processing is done by a Heron Instance that either runs a spout of bolt task and internally uses two threads to receive tuples and gather metrics. Because Heron Instances only run a single process, debugging malfunction tasks becomes easier since logs are written to their own file providing a time order view of events. In case of system overload, Heron uses TCP windowing to propagate back pressure to upstream Heron Instances or spout back pressure to throttle the incoming data flow when Heron Instances start slowing down [45].

D. Samza

Samza [50], similar to Storm and Heron, uses a one-at-a-time continuous processing model and only provides at-least-once semantics. It offers [7] a low-level API comparable to Storm, a SQL-like API for batch processing and unified API to describe stream and batch processing by adopting concepts of the data flow model [13]. Samza itself relies heavily on Kafka [40] for reliable storage, ingestion and replayability of events in case of failing tasks. Samza can either operate on an unbounded data stream (e.g. Kafka topic) or bounded stream (e.g. input files). Independent of the streaming type, a stream is partitioned into an ordered, replayable sequence of records and each message in a partition can be identified given a specific offset [40, 50, 63]. An application in Samza is composed of jobs that contain one or more tasks for processing. In contrast to Storm/Heron, each job is independently deployable and a single-threaded process [18]. Samzas main abstraction is a Job (similar to Storm/Heron Bolt) that executes on data pulled (as a single message item) from Kafka and emit data back to Kafka after processing (as shown in Figure 4).

Similar to Heron, Samza divides the execution plan in a logical and physical representation. The logical plan is a job represented as a directed graph of operators and streams of data. For physical execution and to increase the throughput of the system, a job is divided into multiple parallel, independent, and identical tasks and the input stream is divided into partitions, where each task execute the same logic on its own input partition [50].

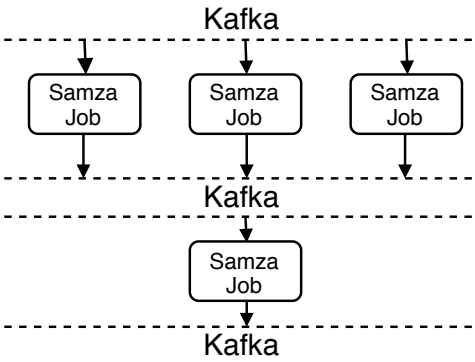

Fig. 4. Samza data flow [63].

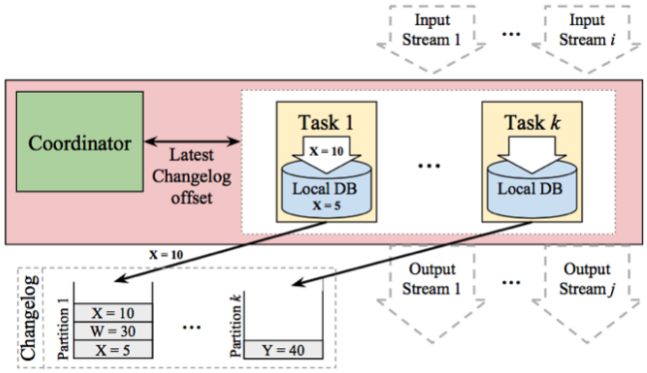

Fig. 5. Samza local state for fault-tolerance [50].

Parallelism can be tuned by running more instances of the Job, but because data is always written back to Kafka between each step in the pipeline, the resulting latency is generally higher compared to systems like Storm. Despite the disadvantages of this approach, it allows for an easy way of sharing data between jobs like combining results of previous stages. Also due to the log semantic, Samza provides ordering of messages across all Kafka partitions [31].

State Management & Fault-Tolerance

Samza guarantees at-least-once processing by giving each task a local, durable state store (key-value store) used to record state and periodically persisting the last processed offset for its input stream partition [7, 50]. As seen in Figure 5 checkpoints are flushed incrementally, only emitting the delta since last checkpoint. They are also replicated to other machines for reliability. On task failure, a new task is spinned up and the old state is restored by replaying the changelog and processing all events after the last offset [7]. Due to the local state, each task is able maintain more data that could fit in memory and because the store is attached directly to each task, lookups are generally faster than remote backends, resulting in lower tail latency and higher throughput. In addition, the enforced buffering of data between processing steps eliminates the need for backpressure since Kafka, given a large scaled deployment, makes sure that the system preserve its stability even if overload occurs. Noghabi et al. also argue that incremental checkpointing is more efficient than full checkpointing considering large state (100TB), but it performs worse than checkpointing in case of smaller state (100GB - 10B), although this is compensable using optimized batching.

Unified API

Instead of using a Lambda Architecture, Samza uses a unified model for batch and stream processing by treating batch as a finite stream and marking the end of the batch with a special token. Samza also splits the processed data into windows (tumpling, session [13]) to avoid reprocessing the entire stream on failures. By default, Samza [7] can reason about processing time of events and provides event-time based processing by its integration with Apache BEAM [2].

When late or out-of-order events arrive, the particular window is re-processed fixing previous results [50].

E. Spark Streaming

Spark Streaming [66] enables stateful stream processing and exactly-once guarantees by using the micro-batch approach, resulting in high throughput, strong consistency guarantees due to the batch mechanism and latencies in the upper milliseconds to seconds. The key concepts of Spark are RDDs and DStreams. RDDs are distributed and immutable collection (resilient distributed datasets) that are manipulated through deterministic operations (transformations) which always produce a new RDD [65]. To shield against process errors or machine failures, Spark keeps track of the full *lineage* of how a RDD evolves (the sequence of operations that created it), checkpointing expensive RDDs so that they can be reused faster in later computation stages [65, 67]. In that way a lost partition can easily be rebuild by replaying saved transformations on the base data, resulting in much faster recovery time compared to syncing state over network from external storage [67]. Streaming computations are expressed as a DStream (discretized streams), a collection of RDDs computed periodically using Spark batch jobs at small time intervals. RDDs are processed in order, whereas the contained data item are processed in parallel, without any ordering guarantee [63]. Figure 6 shows the overall process. In each time interval incoming datasets are stored reliably across the cluster to form an immutable, partitioned dataset [66]. Once the time interval completes, transformations are applied in parallel producing outputs or intermediate results, whereas the latter is stored as a stateful RDD for next batches. If a batch fails, the last RDD can be used to perform the calculation again or if the whole DStream fails, the used dataset is replayable using systems like Kafka.

F. Spark Structured Streaming

Structured Streaming [17] is an improved version of the Spark Streaming extension based on the Spark SQL engine and offers a higher-level API to express batch and stream computations under the same set of API primitives. It offers end-to-end exactly-once semantics and fault-tolerance through

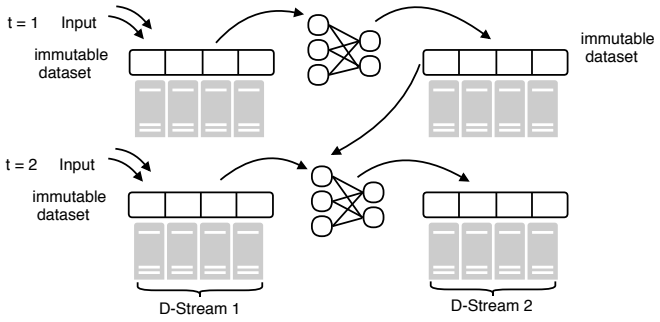

Fig. 6. Spark streaming model [65, 66].

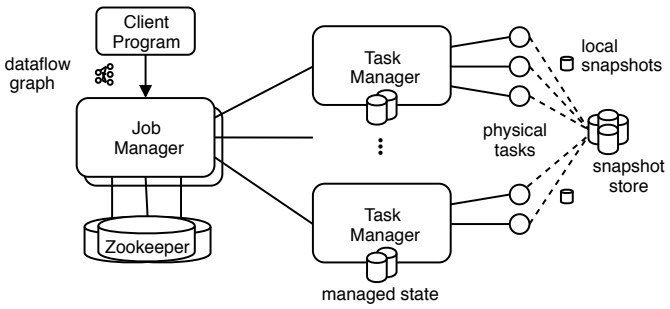

Fig. 7. Flink architecture [22].

checkpointing and write-ahead logs (WAL) [3]. By default, the engine runs in micro-batch mode [66], providing high throughput and exactly-once at the expense of higher latency. In addition, it is able to run long-living operator tasks in continuous processing mode, achieving latencies in the lower milliseconds while softening the guarantees to at-least-once, although exactly-once semantics can be achieved by placing restrictions on sources and sinks [17]. First, the source must be replayable in the case of failure and second, the sink has to handle events idempotently. To maintain state in this mode, Spark asynchronously stores the processed offset to the WAL when reading a so called epoch marker which is injected into the input stream. This allows for periodic state checkpointing without interrupting processing. Like Samza, Spark adopted the concepts of the data flow model [13] and is able to express windowing operators as well as higher-level operations like aggregation or joins on streams. [17].

G. Flink

Flink [4] is a stream-only processing engine that enables low-latency, stateful stream processing, exactly-once semantics and fault-tolerance through asynchronous snapshots [22] which maintain a global state of all operators. Flink adopts the data flow model [13], where batch is only a special case of streaming under fixed window operator to work on a finite dataset. Flink applications are structured as a DAG containing operators which apply transformations onto streams [22]. As core model, Flink uses two abstract data types. *DataStreams* for unbounded streams and *DataSets* for bounded streams. For stateful stream processing, each stream operation in Flink is able to declare its own state to keep track of data seen [22]. The state can reside on heap memory, backed up by a file system or outsourced to an external key-value store (e.g. RockDB) [46]. However, the heap approach has strong limitations in terms of available resources, especially when large aggregates have to be hold in memory.

Figure 7 shows an overview of the Flink architecture, consisting of workers, called TaskManager which are responsible to execute tasks and a JobManager that holds information about active pipelines, coordinates scheduling, observes snapshotting and triggers recovery mechanisms. As seen in Storm and Heron, Flink also relies on Zookeeper [9] to keep metadata, task information and cluster state.

Fault-Tolerance

Based on the famous algorithm developed by Chandy and Lamport, Flinks asynchronous snapshots offers a way to back up pipeline state without stopping the current execution. Therefore each operator persists data between two barrier items that get injected into the data stream. When a barrier is read, the current state is asynchronously persisted to a local database. To obtain a global backup, all local states are pulled into an external data backend, resulting in a global snapshot that can be used to recover the entire pipeline state in the event of an error.

IV. Interesting Topics

Although SPEs have a long list of improvements behind them, there are still a number of open problems to solve and interesting application areas that arise with the new generation of SPEs. Since a detailed consideration of all cases would go beyond the scope of this paper, we will briefly look at the following topics:

1) **benchmarking**: Due to the constantly growing list of SPEs that use a variety of architectural approaches, it is difficult to provide a generic method for evaluating and comparing these systems. Things get even worse because as complexity of these systems increases, small misconfigurations or wrong tuning can lead to a significant change in the benchmark outcome. Hence equal tuning must be applied to the systems under test to get meaningful results [38]. Existing streaming benchmarks often do not cover real-word use cases and provide limited metrics. For example the *Linear Road* [16] benchmark is an application benchmark that simulates a variable tolling system of a metropolitan area for motor vehicle expressways. Besides live data, historical data covering ten weeks of tolling history is generated that need to be combined with the used live queries to produce correct outputs [33]. Linear Road reports a so called L-rating metric which is used to compare the stream data management system with relational databases. *StreamBench* [44] is a benchmark suite that contains several micro-benchmarks (e.g. word count or projection) which focus on benchmarking distributed SPE. Due to its limited scope, it is well suited for

operator comparisons since only relevant parts of the system under test are used. Being a micro benchmark, it has generally limited validity, making it inappropriate to evaluate performance for real-world applications [33]. The queries used in StreamBench vary from simple stateless filter or extraction tasks to complex stateful multi-step queries. However, windowing functions are not included. As input data, StreamBench uses real-word datasets as seed for a more sophisticated data generation and therefore does not work with a complete representation of reality. To cope with generated messages, a broker like Kafka [40] is used to separate generation from consumption. Instead of a self-defined metric, StreamBench measures throughput and latency as well as the penalties resulting from node failures [44]. Ivanov et al. proposed an extension of the well know BigBench for adding stream processing support to the existing big data storage and batch processing benchmarking capabilities. To achieve an end-to-end application benchmark, they outlined a benchmark that has configurable streaming modes for real-time or historic processing use cases. However, it lacks the ability to test windowing, out-of-order, and reprocessing and the supported queries are limited to simple applications. Other notable (but not exhaustive) benchmarks are SparkBench [43], which uses real Twitter data to test the Spark engine, RIotBench [52], a benchmark for SPE at the edge hosting IoT applications and yahoo benchmark [26], which was a first attempt to measure the differences between next generation SPEs like Storm or Flink. It simulates an advertisement analytical pipeline where the SPE's task is to read data provided by Kafka, identify relevant events and store the windowed count of the results to a Redis deployment. In summary, the landscape of up-to-date stream processing benchmarks that are able to cope with the rapid movement of today's SPEs is limited. Only some of them consider the distribution in the exposed metrics and are able to provide a sufficient end-to-end application benchmark for real-word scenarios. While there have been good proposal for this topic [33, 35, 38], there is still a high demand for end-to-end application-level streaming benchmarks with good tooling support.

2) **streaming machine learning:** In the batch-oriented way, machine learning algorithm train a model on large given dataset and the resulting model is used for classifications or predictions. When looking at streaming data, machine learning algorithm cannot iterate over a complete dataset, but must be dynamically (incrementally) updated *online*. In general, streaming or online machine learning covers algorithms that are distributed and have limited historical data access to correct or evaluate decisions made. Therefore they must be able to work on fresh data only. This essentially means that the model must learn a concept step by step by processing labeled training examples one after the other [62].
   In addition, there are a number of requirements and challenges for streaming learning. In order to cope with the data volume, calculations must be distributed and thus need to be implemented using a distributed stream processing system which comes with its own complexity [20]. Since the processing depends on examples generated from the continuous, *non-stationary* flow of data, the model must be adaptable to overcome potential *conceptual drifts* [62].
   Once the above problems have been solved, an inevitable question immediately arises. How does distributed, parallel machine learning work? One way to do parallel learning is to use a parameter server [54] that allows access to shared parameters as key/value pairs. Thereby each training point is split into its constituting attributes and each attribute is sent to a different processing element [20]. Using this mechanism, a linear model trained via gradient descent, for example, can be implemented by storing and updating the coefficients in a distributed store [42]. Popular frameworks that support parallel model training are TensorFlow [29] or MXNet [25]. In the field of distributed stream processing systems, Spark [3] and Flink [4] offer functionality for machine learning algorithms and distributed learning. Spark with its Spark MLLib provides a rich set of batch-oriented machine learning algorithms, but has a limited number of algorithms that take streaming data into account. However Spark has support for several parameter server implementations. A parameter server with recent activity is Angel [36, 37] that has support for linear regression, support-vector-machines and more. In contrast to Spark, Flinks MMLib is less extensive, but also supports model-parallel machine learning with its own implementation of a parameter server. Although there is a adequate support for machine learning, SPE are not specifically tailored towards parallel and scalable incremental model training and inference on event streams [48]. Thus this area of research need more investment. Despite using pure online training approaches, combining batch and online training is also a valuable approach in practice by training a model on precompiled data sets and applying prediction to live data online [20].

3) **performance optimization:** In the past there have been attempts to summarize issues related to performance [34] optimization in the field of stream processing. Due to special use cases and tuning needs, their is still a fair investment in optimizing various aspects of large-scale DSPE deployments. There is a suit of recent papers [57, 61, 68] addressing things like scheduling or processing algorithms, while others evaluate the performance impact of processing guarantees across SPE [11]. In addition, with the increasing number of low-level tuning options and complex streaming operations such as advanced windowing, it is necessary to improve usability, transparency and debugging to cope with the growing complexity.

4) **streaming graph processing:** With real-world graphs such as a social media graph of Twitter interactions ranging the size of millions to even billions of nodes and edges, there is still an interest in handling massive streaming graph deployments [49]. The main areas of focus are concerned with implementing and improving graph streaming algorithms, efficient graph partitioning (process of dividing a graph into a predefined number of subgraphs for distributed computing) and streaming graph processing on linked data like RDF stream processing [27].

V. Conclusion

In this paper, we have looked at the state of the art of distributed stream processing engines by analyzing their architectures, state handling and processing guarantees.

Before that, we devoted the first chapter to the differences between batch and stream processing and the challenges of processing events at-motion, which resulted in the development of two widely used architectures. While the Lambda Architecture was designed to compensate for the approximate results of streaming layers by effectively running the calculation logic twice, the next generation DSPE features high performance and throughput characteristics as well as strong processing guarantees for stateful workloads. This led to the idea of the Kappa Architecture which treats streaming as a first-class citizen and uses systems like Kafka to have a reliable, scalable and most importantly repeatable streaming data source. Although Lambda and Kappa Architecture had their moments in the early days of immature DSPEs, the distinction between them will fade as DSPEs become more advanced and powerful. Projects like Apache BEAM continue to support this development.

After glancing over the basics, we focused on the analysis of current SPEs. In general, these systems can be divided into two types. Systems like Apache Spark and Apache Trident use the micro-batch model, trading latency for throughput. In contrast, Apache Storm, Apache Heron, Apache Samza and Apache Flink use the continuous operator model that uses an operator graph of long-running tasks and process events on-at-a-time. In order to achieve fault-tolerant execution, the above systems use a variety of mechanisms that are directly related to their processing guarantees. While Storm/Heron achieve at-least-once through per-record acknowledgement and fault-tolerance by synchronously backing up processing state to an external key-value store, Spark Streaming is able to provide exactly-once semantics based on the discretized streams and RDD abstraction in combination with barrier synchronization to checkpoint expensive calculations.

Samza embraces the idea of local state and continuously checkpoints the state differences into a changelog instead of communicating with a remote storage system. Flink also provides exactly-once by using local state and global asynchronous snapshotting to be able to recover processing on failures. This pushes the complexity outside the system, relying on high-performance key-value stores and optimized network infrastructure. Consequently using this approach for recovering is expensive, because the state for all operators must be reset, resuming computation to the last known checkpoint.

Lastly, we have shown that despite the fast evolving landscape of DSPE, there are still open challenges. In particular benchmarking and performance evaluation is a ubiquitous problem to make the different DSPE comparably and help in the decision making process. Existing benchmark suits are either not directly designed towards streaming analysis, offer limited queries and metrics or do not adequately reflect real use cases. Moreover, we briefly discussed the area of distributed machine learning in combination with DSPEs using a distributed parameter server and identified some areas of research in the field of stream graph processing.

VI. Further Work and Personal Outlook

The goal of this paper was to lay out the current state of the art for distributed stream processing engines. I got a good overview of the underlying procedures and trade-offs that led to the individual design decisions. Nonetheless it is inevitable for my further journey to develop practical knowledge in stream processing by understanding the operations done on streams [60] and to actually use a DSPE in a production-like scenario. In addition, the research made clear that the topic requires deeper knowledge of the way these system are built to give a sophisticated opinion.

During the base project I would like build a stream processing architecture using a subset of the presented SPE in cooperation with the *HAW Big Data Lab* and further evaluate the benchmarking and streaming machine learning applications, for which Benczúr et al. provided a good starting point. I am also open for the streaming graph processing area, but this field will require more knowledge in the currently used algorithms and techniques for handling large-scale streaming graphs.

Ideally, the basic project leads to a specialization on one of the selected topics and results in the concrete research question for my master thesis.

References

[1] Apache Apex, . https://apex.apache.org/.

[2] Apache Beam, . https://beam.apache.org/.

[3] Apache Spark, . https://spark.apache.org/.

[4] Apache Flink. https://flink.apache.org/.

[5] Apache Heron. https://apache.github.io/incubator-heron/.

[6] Apache Kafka. https://kafka.apache.org/.

[7] Apache Samza. https://samza.apache.org/.

[8] Apache Storm. http://storm.apache.org/.

[9] Apache Zookeeper. https://zookeeper.apache.org/.

[10] D. J. Abadi, Y. Ahmad, M. Balazinska, U. etintemel, M. Cherniack, J.-H. Hwang, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[11] S. M. A. Akber, C. Lin, H. Chen, F. Zhang, and H. Jin. Exploring the impact of processing guarantees on performance of stream data processing. In *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, pages 1286–1290, 2017.

[12] T. Akidau and A. o. Balikov. Millwheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, Aug. 2013.

[13] T. Akidau, R. Bradshaw, C. Chambers, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Aug. 2015.

[14] A. Anthony. Observability at Twitter: technical overview, part I. 2016. URL https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html.

[15] A. Arasu, B. Babcock, S. Babu, et al. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 2003.

[16] A. Arasu, M. Cherniack, M. Stonebraker, et al. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491, 2004.

[17] M. Armbrust, T. Das, M. Zaharia, et al. Structured streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613. ACM, 2018.

[18] M. D. Assuncao, A. D. S. Veith, and R. Buyya. Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. 2017.

[19] H. Balakrishnan, M. Balazinska, M. Stonebraker, et al. Retrospective on Aurora. *The VLDB Journal*, pages 370–383, 2004.

[20] A. A. Benczúr, L. Kocsis, and R. Pálovics. Online Machine Learning in Big Data Streams. *CoRR*, 2018.

[21] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proc. VLDB Endow.*, pages 1441–1451, 2014.

[22] P. Carbone, S. Ewen, et al. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.*, pages 1718–1729, 2017.

[23] S. Chandrasekaran, O. Cooper, A. Deshpande, et al. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[24] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, pages 63–75, 1985.

[25] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, 2015.

[26] S. Chintapalli, D. Dagit, et al. Benchmarking Streaming Computation Engines at Yahoo!, Dec 2015. URL https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at.

[27] M. Dayarathna and S. Perera. Recent Advancements in Event Processing. *ACM Comput. Surv.*, pages 33:1–33:36, 2018.

[28] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, pages 107–113, 2008.

[29] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, 2012.

[30] M. N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: you only get one look a tutorial. In *SIGMOD Conference*, 2002.

[31] F. Gessert. Scalable Stream Processing: A Survey of Storm, Samza, Spark and Flink by Felix Gessert, 2017. URL youtube.com/watch?v=ZWez6hOpirY.

[32] M. Grover, T. Malaska, J. Seidman, and G. Shapira. *Hadoop Application Architectures*. O'Reilly Media, Inc., 1st edition, 2015.

[33] G. Hesse, C. Matthies, B. Reissaus, and M. Uflacker. A New Application Benchmark for Data Stream Processing Architectures in an Enterprise Context: Doctoral Symposium. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 359–362, New York, NY, USA, 2017. ACM.

[34] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.*, pages 46:1–46:34, 2014.

[35] T. Ivanov, P. Bedué, et al. Adding Velocity to BigBench. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, pages 6:1–6:6, New York, NY, USA, 2018. ACM.

[36] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of*

the 2017 ACM International Conference on Management of Data, SIGMOD '17, pages 463–478. ACM, 2017.

[37] J. Jiang, L. Yu, B. Cui, J. Jiang, and Y. Liu. Angel: a new large-scale machine learning system. *National Science Review*, pages 216–236, 2017.

[38] J. Karimov, T. Rabl, A. Katsifodimos, et al. Benchmarking Distributed Stream Processing Engines. *CoRR*, 2018.

[39] J. Kreps. Questioning the lambda architecture, 2014. URL https://streaml.io/blog/exactly-once.

[40] J. Kreps, N. Narkhede, and J. Rao. Kafka : a Distributed Messaging System for Log Processing. 2011.

[41] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[42] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, et al. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, 2014. USENIX Association.

[43] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 53:1–53:8, New York, NY, USA, 2015. ACM.

[44] R. Lu, G. Wu, B. Xie, and J. Hu. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 69–78, Washington, DC, USA, 2014. IEEE Computer Society.

[45] F. Maosong, M. Sailesh, et al. Streaming@twitter. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 2015.

[46] O. Marcu, R. Tudoran, B. Nicolae, et al. Exploring Shared State in Key-Value Store for Window-Based Multi-pattern Streaming Analytics. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 1044–1052, 2017.

[47] N. Marz. How to beat the CAP theorem, 2011. URL http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[48] C. Mayer, R. Mayer, and M. Abdo. Streamlearner: Distributed Incremental Machine Learning on Event Streams: Grand Challenge. *CoRR*, 2017.

[49] A. McGregor. Graph Stream Algorithms: A survey. *SIGMOD Rec.*, pages 9–20, 2014.

[50] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.*, pages 1634–1645, 2017.

[51] J. Peng. Exactly once is NOT exactly the same. URL https://streaml.io/blog/exactly-once.

[52] A. Shukla, S. Chaturvedi, and Y. Simmhan. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing platforms. *CoRR*, 2017.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[54] A. Smola and S. Narayanamurthy. An Architecture for Parallel Topic Models. *Proc. VLDB Endow.*, pages 703–710, 2010.

[55] Steven Wu. Steven Wu, Netflix—Flink Forward 2018, 2018. URL https://www.youtube.com/watch?-v=dMvHQ0TPWBY.

[56] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, pages 42–47, 2005.

[57] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 66–77, New York, NY, USA, 2017. ACM.

[58] Q. To, J. Soto, and V. Markl. A Survey of State Management in Big Data Processing Systems. *CoRR*, 2017.

[59] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, et al. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[60] A. Tyler, C. Slava, and R. Lax. *Streaming Systems - The What, Where, When, and How of Large-Scale Data Processing*. 2018.

[61] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[62] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, pages 69–101, 1996.

[63] W. Wolfram, G. Felix, F. Steffen, and R. Norbert. Real-time stream processing for Big Data. *it - Information Technology*, pages 186–194, 2016.

[64] A. Woodie. Kafka tops 1 trillion messages per day at linkedin, 2015. URL https://www.datanami.com/-2015/09/02/kafka-tops-1-trillion-messages-per-day-at-linkedin/.

[65] M. Zaharia, M. Chowdhury, T. Das, et al. Resilient Distributed Datasets: Aa Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design*

and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[66] M. Zaharia, T. Das, H. Li, et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[67] M. Zaharia, R. S. Xin, P. Wendell, et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, pages 56–65, 2016.

[68] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: Optimized Shuffle Service for Large-scale Data Analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.