Dagstuhl Seminar on Big Stream Processing

Sherif Sakr
Uni. of Tartu, Estonia
sherif.sakr@ut.ee

Tilmann Rabl
TU Berlin, Germany
rabl@tu-berlin.de

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Paris Carbone
KTH EECS, Sweden
parisc@kth.se

Martin Strohbach
AGT International, Germany
mstrohbach@agtinternational.com

ABSTRACT

Stream processing can generate insights from big data in real time as it is being produced. This paper reports findings from a 2017 seminar on big stream processing, focusing on applications, systems, and languages.

1. OVERVIEW

As the world gets more instrumented and connected, we are witnessing a flood of raw data generated, at high velocity, from different hardware (e.g., sensors) or software in the form of streams of data. Examples abound in several domains including financial markets, surveillance systems, manufacturing, smart cities, and scalable monitoring infrastructure. In these domains, there is a strong requirement to collect, process, and analyze big streams of data to extract valuable information, discover new insights in real-time, and detect emerging patterns and outliers. Since 2011 alone, several systems (e.g., SPL [13], Storm [19], Apex[1], Spark Streaming [20], Flink [7], Heron [16], and Beam [3]) have been introduced to tackle the real-time processing demands of big streaming data. However, there are several challenges and open problems that need to be addressed to improve the state-of-the-art and achieve further adoption of big stream processing technology [18].

This report is based on a seminar on "*Big Stream Processing Systems*" at Schloss Dagstuhl in Germany from 29 October to 3 November 2017[2], attended by 29 researchers from 13 countries. Participants came from different communities including systems, query languages, benchmarking, stream mining, and semantic stream processing. A benefit of this seminar was the opportunity for scholars from different communities to get exposure to each other and get freely engaged in direct and interactive discussions. The program consisted of tutorials on the main topics of the seminar, lightning talks by participants on their research, and two working groups dedicated to a deeper investigation. The first working group focused on applications and system of big stream processing while the second group focused on streaming languages. This report presents highlights and outcomes.

[1] https://apex.apache.org/

[2] www.dagstuhl.de/en/program/calendar/semhp/?semnr=17441

2. TUTORIALS

The tutorials of the seminar aimed at sharing knowledge between attendees from different communities, offering perspectives for group discussions.

2.1 IoT Stream Processing Applications

This tutorial analyzed IoT applications from two domains: sports and entertainment as well as Industry 4.0. The application examples are based on commercial deployments using AGT International's[3] Internet of Things Analytics (IoTA) platform.

Sports and Entertainment. The example applications of this domain provide real-time narratives about highlights during a live event. This way, it is not necessary to watch the whole event, but one can be notified in real-time about such highlights based on insights derived from sensor data. For instance, in *basketball*, sensors that have been successfully used in commercial deployments[4] include smart shirts worn by players, microphones deployed to monitor the audience, cameras, and wristbands. Data from these sensors in combination with play-by-play data can be used to recognize behavior, emotions, activities, actions, pressure, and other physical aspects of the game. These insights are related to players, teams, fans, and family preferably in the form of semantic data streams. Semantic data access decouples applications from data providers and enables domain experts to better work with the data, e.g., for generating content and distributing it via social media.

Another example is *mixed martial arts*[5], where cameras and sensors embedded in floors and fight-

[3] http://www.agtinternational.com

[4] https://t.co/ZkQjQwXw13

[5] https://youtu.be/vataVq9gY_o

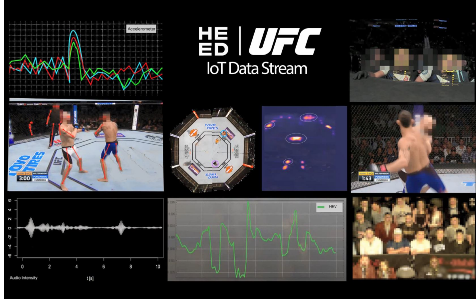

Figure 1: Sample IoT data streams in mixed martial arts.

ers' gloves[6] offer insights including punch strength and stress levels of each fighter (Figure 1). In this example, it is important that insights can be delivered in real-time without noticeable delay compared to a broadcast of the fight.

In *professional bull riding*, sensors are attached to riders and bulls and used to quantify the bull's and rider's performance[7]. As this information is, among other things, used for automatic scoring, it is of particular importance that analytic results are available as soon as the ride is finished. Similarly, a range of wearable sensors are used for creating event highlights for participants at *mass sport events* such as the Color Run[8]. The CPaaS.io project[9] uses action cameras and fitness bands to automatically detect event highlights based on the the runner's activity, emotions, dance energy levels, and many more metrics. In this application, real-time aspects include scenarios in which event highlights are being directly sent to friends of the participants.

Industry 4.0. For this domain, the tutorial presented applications around *predicting energy peaks* and *predictive maintenance*. In principle, predicting energy peaks can help in reducing energy costs as electricity bills of industrial consumers contain a pricing component that incurs higher charges for higher peaks of electrical load. For small-to-medium enterprises, avoiding such peak load events can lead to significant savings[10]. This can be achieved by predicting expected peaks, e.g., up to 30 minutes ahead of time and taking precautionary measures such as temporarily switching off high energy consumers such as air conditioning.

For *predictive maintenance*, the tutorial presented an application for detecting anomalous machine states to reduce maintenance costs. For instance, in injection molding machines, a sudden high energy consumption may indicate that an injection nozzle is jammed and checking the machine may avoid further damage. The tutorial reported about the DEBS Grand Challenge 2017 [11] that has been designed to objectively measure some of these requirements using pre-defined machine learning algorithms and RDF streaming data. The main KPI for the challenge was latency. The original data set has been provided by Weidmüller[11]. For reasons of confidentiality, the organizers provided a mimicked data set[12]. The systems under test were evaluated using the HOBBIT benchmarking platform[13] that ensured the objectivity of quantifying the performance of distributed stream processing pipelines. Overall, 7 out of 14 participating teams in the challenge passed the correctness test. The fastest system [4] achieved an average latency of about 39ms. The DEBS Grand Challenge 2017 benchmark is openly available as part of the HOBBIT platform.

2.2 Big Stream Processing Systems

This tutorial started by identifying the most differentiating characteristic of scalable data stream processing systems, which is the notion of data as a continuous, possibly infinite resource instead of "facts and statistics organized and collected together for future reference or analysis"[14]. In fact, data stream processing systems broaden the context from retrospective data analysis to continuous, unbounded processing coupled with scalable and persistent application state. Various forms of stream processing have been employed in the past within their respective domains, such as network-centric processing on byte streams, functional (e.g., monads) and actor programming, complex event processing, and database materialized views. Besides, *stream management* has been an active research field for many years [2, 5, 8]. Nonetheless, several of these ideas have only just recently been put together in a consistent manner to compose a stack centered around the notion of data as an unbounded partitioned stream of records (Figure 2). Most importantly, stream processing did not restrict but complemented existing scalable processing models (e.g., MapReduce [10]) with persistent partitioned state, time domains, and flexible scoping via windows. The general programming stack addresses storage, compute, and domain-specific library support.

Stream Storage. Data dissemination from consumers to producers is a problem that has been revisited multiple times with different assumptions and needs in mind. In the context of data streaming, direct communication (e.g., TCP channels) was not an option despite low-latency requirements, since it required application ingestion to be actively in

[6] http://bit.ly/2D4lCqD
[7] http://bit.ly/2CXpc2g
[8] https://thecolorrun.com/
[9] http://www.cpaas.io
[10] http://bit.ly/2DjhvUh
[11] http://www.weidmueller.de
[12] https://hobbit.iminds.be/dataset/weidmuller
[13] http://bit.ly/2muMNkY
[14] Google Dictionary

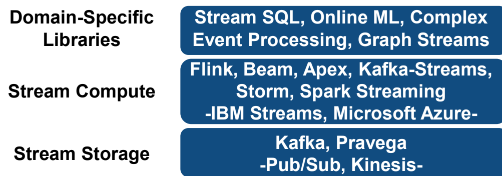

Figure 2: The Stack of Scalable Stream Processing

sync with data creation while also lacking the transparency and durability of today's cloud computing ecosystem. Furthermore, message brokers (e.g., RabbitMQ, JMS) were insufficient for the needs of supporting multiple applications and configurations (i.e., task parallelism). Thus, a class of open-source stream storage systems based on *partitioned replicated logs* was introduced, led by Apache Kafka [15] and more recently Pravega[15] as well as proprietary cloud services such as Amazon Kinesis[16]. Partitioned replicated logs provide high sequential read and write throughput by exploiting copy-on-write and strict data-parallel access by distinct consumers. Furthermore, they perform offset-based bookkeeping of data access for the purposes of data reprocessing, reconfiguration, and roll-backs, among others. Finally, more effort has been devoted to supporting transactional logging and repartitioning, allowing for seamless integration with modern stream compute systems.

Stream Compute. We further divide compute into *programming models* and *runtime engines*. In terms of programming model support, there has been a shift from purely event-based, compositional models (e.g., Apache Storm [19]) to more declarative representations [3, 7, 20]. Currently, most standard APIs are fluid, functional, and allow declaring relational transformations (e.g., joins, filters) while providing first-class support for persistent partitioned state, stream windows, and event-time progress using watermarks. The latter allowed application logic to incorporate timers that operate consistently on different time domains (e.g., origin-time), thus allowing out-of-order processing [17], a concept popularized e.g. by Beam [3].

With respect to runtime engines, we observe converging commonalities such as a dataflow execution model, explicit locally embedded state (using log-compaction trees [1]), and asynchronous snapshots for fault tolerance and reconfiguration [6, 14]. Spark Streaming [20], as a special case, emulates streaming by slicing computation into recurring batch jobs, yet, it currently makes use of locally embedded state and there are plans to adopt a continuous processing runtime for low-latency data streaming.

[15] http://pravega.io/ [16] https://aws.amazon.com/kinesis/

2.3 Stream Processing Languages

This tutorial provided an overview of several styles of stream processing languages: streaming SQL, synchronous dataflow, big-data streaming, complex event processing, and end-user programming. After the Dagstuhl seminar, some of the participants wrote a survey paper inspired by this tutorial [12]. For space reasons, rather than describing the tutorial here, we refer interested readers to that paper.

3. WORKING GROUPS

During the seminar, two separate working groups formed to discuss current challenges in streaming applications and systems and in streaming languages.

3.1 Applications and Systems

In this working group, participants discussed characteristics and open challenges of stream processing systems, focusing on state management, transactions, and pushing computation to the edge.

State Management. Modern streaming systems are stateful, which means they can remember the state of the stream to some extent. A simple example is a counting operator that counts the number of elements seen so far. While even a simple state like this poses several challenges in streaming setups (such as fault tolerance and consistency), many use cases require more advanced state management. An example is the combination of streaming and batch data, e.g., when combining the history of a user with their current activity or when finding matching advertisement campaigns with current activity; a popular example of such a setup is modeled in the Yahoo! Streaming Benchmark [9]. Today, most setups deal with such challenges by combining different systems (e.g., a key value store for state and a streaming system for processing). However, it is desirable to have both in a single system for consistency and manageability reasons.

State can be considered the equivalent of a table in a database system [5]. As a result, several high-level operations can be identified: conversion of streams to tables (e.g., storing a stream), conversion of tables to streams (e.g., scanning a table), as well operations only on tables or streams (joins, filters, etc.). The management of state opens the design space between existing stream processing systems and database systems, which has only been partially explored by current systems. In contrast to database systems, stream systems typically operate in a reactive manner, i.e., they have no control over the incoming data stream, specifically, they do not control and define the consistency and order

semantics in the stream. This requires advanced notions of time and order as for example specified for streams in the dataflow model [3].

Transactions. A further discussion topic was transactions in stream processing systems. The main difference between traditional database transactions and stream processing transactions is that in databases the computation moves and data stays (in the system), whereas in stream processing systems the computation stays and the data moves to the computation (and out again). Considering state management, the form of transactions as applied in databases can also be used in a stream processing system, if the state is managed in a transactional way. However, the operations on streams themselves can be transactional and then we can differentiate between single-tuple transactions and multi-tuple transactions (possibly accessing multiple keys in a partitioned operator state space). Multi-tuple transactions can only commit when all tuples are consumed. The tuples then have to traverse the whole operator graph or at least the transactional subgraph. The semantics of transactions on streams is currently still an open field of research.

Pushing computation to the edge of a network enables stream processing to be highly distributed and decentralized. This is very useful when preprocessing or filtering can be done without a centralized view of the data, especially in setups with high communication cost or slow connections (e.g., mobile connections): it makes sense to not send all data to a central server, but distribute the computation. A logical first step is filtering, but aggregations and even more complex operations can be pushed to the edge, if possible. Many modern scenarios prohibit centralized data storage, which further encourages distributed setups with early aggregations.

3.2 Languages and Abstractions

Based on the corresponding tutorial (Section 2.3), this working group identified three challenges faced by streaming languages: input variety, output veracity, and adoption of streaming languages. After the seminar, some of the participants continued the discussion and incorporated it in the same survey paper that was inspired by the tutorial [12].

4. CONCLUSION

The tutorials, presentations, dialogs, and working groups at the "*Big Stream Processing Systems*" seminar provided an overview of current developments and emerging issues. This report highlighted the main outcomes of the seminar. The discussions of the seminar have also revealed several open challenges and interesting future research directions including (1) semantic data access and reasoning, (2) defining a standardized query language for streaming applications, (3) providing better support for machine learning including a wide range of data science programming languages (Python, R, Julia), and (4) improving optimizations for low latencies and short-lived stream processing pipelines.

Acknowledgements

We thank the seminar participants and the Dagstuhl staff. The work presented in this paper has partly been funded by the European Regional Development Fund via the Mobilitas Pluss program (grant MOBTT75), by the H2020 projects CPaas.io, HOBBIT, Streamline, and Proteus (grant agreement numbers 688227, 723076, 688191, and 687691), and by the German Ministry for Education and Research as Berlin Big Data Center (funding mark 01IS14013A).

5. REFERENCES

[1] Rocksdb. http://rocksdb.org/, 2018.
[2] D. J. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB J.*, 12(2), 2003.
[3] 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. In *VLDB*, pages 1792–1803, 2015.
[4] C. Amariei, P. Diac, and E. Onica. Optimized stage processing for anomaly detection on numerical data streams: Grand challenge. In *DEBS*, 2017.
[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
[6] P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. In *VLDB*, 2017.
[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Database Engineering Bulletin*, 36(4):28–38, 2015.
[8] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 668–668, 2003.
[9] S. Chintapalli et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *IPDPSW*, 2016.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *CACM*, 51(1), 2008.
[11] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow. The DEBS 2017 grand challenge. In *DEBS*, 2017.
[12] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Vlachou. Stream processing languages in the big data era. *SIGMOD Record*, 2018.
[13] M. Hirzel, S. Schneider, and B. Gedik. SPL: An extensible language for distributed stream processing. *Transactions on Programming Languages and Systems (TOPLAS)*, 39(1):5:1–5:39, 2017.
[14] G. Jacques-Silva et al. Consistent regions: Guaranteed tuple processing in ibm streams. In *VLDB*, 2016.
[15] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. NetDB, 2011.
[16] S. Kulkarni et al. Twitter Heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.
[17] J. Li et al. Out-of-order processing: A new architecture for high-performance stream systems. In *VLDB*, 2008.
[18] S. Sakr. *Big Data 2.0 Processing Systems: A Survey.* Springer, 2016.
[19] A. Toshniwal et al. Storm @Twitter. In *SIGMOD*, 2014.
[20] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.