Dagstuhl Seminar on Big Stream Processing

Sherif Sakr
KSAU-HS & UNSW
ssakr@cse.unsw.edu.au

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

Martin Hirzel
IBM Research AI
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 that is getting generated, at high velocity, from different hardware (e.g., sensors) or software in the form of streams of data. Examples of this phenomenon are crucial for several applications and domains including financial markets, surveillance systems, manufacturing, smart cities, and scalable monitoring infrastructure. In these applications and domains, there is a strong requirement to collect, process, and analyze big streams of data in order to extract valuable information, discover new insights in real-time, and detect emerging patterns and outliers. Since 2011 alone, several systems (e.g., Storm [31], Apex[1], Spark Streaming [33], Flink [12], and Heron [26]) 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 in order to improve the state-of-the-art in this domain and achieve further adoption of big stream processing technology [29].

This report is based on a five-day seminar on "*Big Stream Processing Systems*" that took place at Schloss Dagstuhl in Germany from 29 October to 3 November 2017[2]. The seminar was attended by 29 researchers from 13 countries. Participants came from different communities including systems, query languages, benchmarking, stream mining, and semantic stream processing. A major 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 seminar 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 of selected topics. The first group focused on applications and system of big stream processing while the second group focused on analyzing the state-of-the-art of stream processing languages. This report presents highlights and outcomes.

2. TUTORIALS

The tutorials of the seminar aimed at sharing knowledge between attendees from different communities, thus offering insights and perspectives that enriched the group discussions. This section gives an overview of the material presented at tutorials.

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 that are happening 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

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

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

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

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

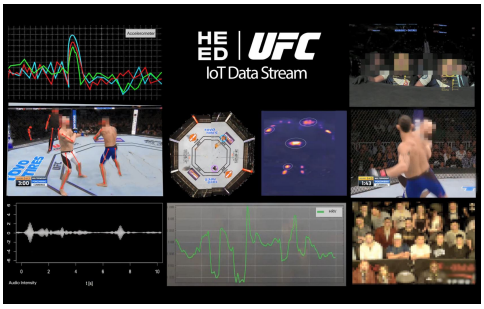

Figure 1: Sample IoT data streams in mixed martial arts.

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], in which sensors such as cameras, smart floors, and sensors embedded in fighters' gloves[6] are used to determine a range of 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]. In the CPaaS.io project[9], an application has been developed to use 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 in order 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 [19] 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 [5] 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, 7, 13]. Nonetheless, several of these

[5] https://youtu.be/vataVq9gY_o
[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] Definition of "data" according to Google Dictionary

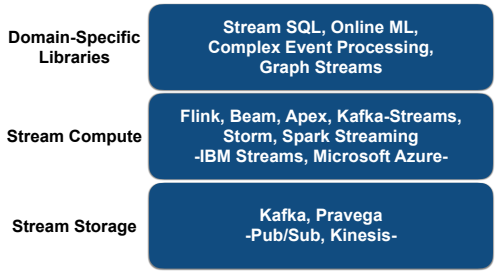

Figure 2: The Stack of Scalable Stream Processing

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 [15]) 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 sync with data creation while also lacking the transparency and durability properties that are the norm in 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 [25] 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 [31]) to more declarative representations [12, 4, 33]. 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 [28], a concept popularized, among others, by Google [3, 4].

With respect to runtime engines, we observe certain converging commonalities such as a dataflow execution model, explicit locally embedded state (using log-compaction trees [1]), and asynchronous state snapshotting for fault tolerance and reconfiguration support [11, 24]. Spark Streaming [33], as a special case, emulates data 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 as well for the purposes of low-latency data streaming.

2.3 Stream Processing Languages

This tutorial provided an overview of several styles of stream processing languages. The tutorial illustrated each style (e.g., relational, synchronous) with a representative example language. Of course, for each style, there is an entire family of languages, and this tutorial did not aim to be exhaustive.

In general, a *stream* is a conceptually infinite ordered sequence of data items, and a *streaming application* is a computer program that continuously ingests input streams and produces output streams. A *stream processing language* is a DSL (domain-specific language) for writing streaming applications. For clarity, this tutorial focused on stand-alone DSLs, as opposed to embedded DSLs [23] that build on a general-purpose host language to obviate the need for a dedicated compiler.

Streaming SQL dialects are an attempt to be for streaming data what SQL has been for data stored in a database. A prominent example is CQL [6], which extends the familiar select-from-where syntax of SQL with windows that turn the recent history of a stream into a relation, as well as with constructs that watch the changes happening to a relation over time and derive a stream from them. CQL is implemented via translation into a streaming extension of relational algebra.

Synchronous Dataflow languages offer streaming with deterministic concurrency for reliable embed-

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

ded control systems. An example is StreamIt [30], where streaming applications are graphs composed of only four constructs: individual operators, pipelines of operators, feedback loops, and split-merge topologies that implement task parallelism. The StreamIt compiler determines a repeating steady-state schedule, then exploits that for optimizations such as operator fusion and data parallelism.

Big-Data Streaming languages focus on large-scale stream processing applications with a variety of data and processing requirements. An example is SPL [22] which offers parallelism across both multi-core machines and multi-machine clusters. SPL addresses a variety of data requirements via rich data types and assorted parsing operators. And SPL addresses a variety of processing requirements by letting programmers write new first-class streaming operators in their language of choice.

Complex Event Processing (CEP) patterns describe how to detect high-level events from a sequence of low-level events in a stream. CEP is usually implemented via some form of automaton. Given that regular expressions are the most popular surface language for automatons, MatchRegex uses regular expressions for patterns over streams [20]. Low-level events are detected via simple predicates. In addition, aggregation functions over events serve to both guide and summarize matches.

Reactive Programming languages specify computation that depends upon variables whose values change subject to streaming data, and propagate any updates to the result of the computation. Reactive programming is analogous to the behavior of spreadsheet formulas, and ActiveSheets takes that analogy to its logical conclusion [21]. Besides combining spreadsheets and streams, ActiveSheets also augments them with time-based windows, key-based partitioning, and an optimized implementation.

Controlled Natural Languages (CNLs) are artificial languages (such as programming languages) designed to look like natural languages (such as English). The META platform provides a CNL for event-condition-action rules over event streams [8]. CNLs can be read and understood by any speakers of the corresponding natural language even without technical training required by normal programming languages. Using a CNL for streaming therefore contributes to the democratization of streaming.

3. WORKING GROUPS

During the seminar, two separate working groups formed to discuss current challenges in the topics "stream processing applications and systems" and "stream processing languages".

3.1 Applications and Systems

In this working group, participants discussed characteristics and open challenges of stream processing systems. The discussions mainly focused on the topics of 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 capabilities. An example is the combination of streaming and batch data. This is for example required when combining the history of a user with her 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 [14]. 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 [6]. As a result, besides the combination of stream and state, 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 [4]; the combination of stream and state, this remains an open field of research.

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 definitions and survey from the corresponding tutorial (see Section 2.3), this working group identified and described three challenges faced by stream processing languages.

Variety of Data Models. A data model organizes elements of data with respect to their semantics, their logical composition into data structures, and their physical representation. Producers and consumers of streams to and from a streaming application dictate data models it must handle, and the application's own conversion and processing needs drive additional data-model variety. There is no consensus on the definition of a stream data item. At one extreme, in Streamlt, each data item is a simple number [30], while at the other extreme, C-SPARQL streams entire self-describing graphs [10]. Streaming languages have so far failed to consolidate on a unified data model because data-model variety is a difficult challenge.

Data-model variety causes streaming-specific issues, since the data model affects the speed of serialization, transmission, compression, and dynamic checks for the presence or absence of certain fields, and because the online setting leaves no time for separate batch data integration. Some stream processing languages are designed around their data model, e.g., CQL on tuples [6] or path expressions on XML trees [16]. Furthermore, the data model enables streaming-language compilers to provide helpful error messages and optimizations. The goal is for streaming languages to let the programmer use the logical data model they find most convenient while letting the compiler choose the best physical representation. Metrics of success include the expressive power of the language along with its throughput, latency, and resource consumption.

Veracity with Simplicity. Veracity means producing accurate and factual results, and simplicity means avoiding unnecessary language complexity. There are several reasons why streaming veracity is hard. Sensors producing input data have limited precision, energy, and memory. In long-running and loosely-coupled streaming applications, sources come and go. And approximate streaming algorithms [9] and stream mining [17] introduce additional uncertainty. This is compounded by the lack of ground truth in an online setting and by the difficulty of anticipating and testing every eventuality.

Veracity causes streaming-specific issues, since it requires accurate real-time responses without having seen all the data, and because the online setting leaves no time for separate batch data cleansing. Also, streaming is often used in a distributed setting, where there can be no global clock [27]. Some streaming languages are explorations in handling uncertainty on top of stream-relational algebra [32], but restricting stream operators to support retraction or uncertainty propagation limits expressiveness and raises complexity. A more general solution might use probabilistic programming to handle uncertainty in a principled way [18]. The goal is for streaming languages to help minimize compounding uncertainty by being quality-aware and adaptive while remaining simple, expressive, and fast. This inherently leads to multiple metrics (e.g., precision, recall, throughput, latency) and harder-to-quantify objectives (simplicity, expressiveness). One can maximize one set of metrics while satisficing a threshold on the others, or one can seek Pareto-optimal solutions [34].

Adoption. While there are many languages, none have reached broad acceptance and use. The community should care about the adoption of streaming languages because it would drive adoption of streaming technologies in general. A widely-adopted language is more attractive for students to learn, leading to a bigger pool of skilled people to hire for companies. Furthermore, a widely-adopted lan-

guage would lead to more mature libraries, tools, benchmarks, and optimizations. The lack of a dominant language indicates that adoption is a difficult goal. Streaming as a domain is young, fast-moving, and prone to vendor lock-in. Meanwhile, there is so far neither a consensus on a streaming language nor even a consensus on which language features are the most important and which can be omitted to reduce complexity. Furthermore, several recent streaming systems have a DSEL, which tends to have less well-isolated semantics and more host-language dependencies than a stand-alone DSL. The goal is for the community to agree upon one or a few languages that get widely adopted. Metrics for the adoption of a streaming language include the number of applications written in it, as well as mentions in resumes, job posting, courses, and support forums. Adoption can also be measured by the number of streaming systems that support a language, open-source and open-governance implementations, and ultimately, an industry standard.

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 in the areas of systems, computational models, architectures, and languages for processing large-scale streaming data. This report highlighted the main outcomes of the seminar. The discussions of the seminar have also revealed the existence of several open challenges and interesting future research directions including the focus on (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 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. *VLDBJ*, 2003.
[3] T. Akidau et al. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
[4] 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. *VLDB*, 2015.
[5] C. Amariei, P. Diac, and E. Onica. Optimized stage processing for anomaly detection on numerical data streams: Grand challenge. In *DEBS*, 2017.
[6] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2), 2006.
[7] A. Arasu et al. Stream: The stanford data stream management system. *Book chapter*, 2004.
[8] M. Arnold et al. META: Middleware for events, transactions, and analytics. *IBM Journal of Research & Development*, 60(2–3), 2016.
[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
[10] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *WWW-Poster*, 2009.
[11] P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *PVLDB*, 2017.
[12] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.
[13] S. Chandrasekaran et al. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.
[14] S. Chintapalli et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *IPDPSW*, 2016.
[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1), 2008.
[16] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.
[17] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *SIGMOD Record*, 34(2), 2005.
[18] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *ICSE*, 2014.
[19] V. Gulisano et al. The DEBS 2017 grand challenge. In *DEBS*, 2017.
[20] M. Hirzel. Partition and compose: Parallel complex event processing. In *DEBS*, 2012.
[21] M. Hirzel, R. Rabbah, P. Suter, O. Tardieu, and M. Vaziri. Spreadsheets for stream processing with unbounded windows and partitions. In *DEBS*, 2016.
[22] 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, March 2017.
[23] P. Hudak. Modular domain specific languages and tools. In *ICSR*, 1998.
[24] G. Jacques-Silva et al. Consistent regions: guaranteed tuple processing in ibm streams. *PVLDB*, 9(13), 2016.
[25] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. NetDB, 2011.
[26] S. Kulkarni et al. Twitter Heron: Stream processing at scale. In *SIGMOD*, 2015.
[27] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *CACM*, 21(7), 1978.
[28] J. Li et al. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1), 2008.
[29] S. Sakr. *Big Data 2.0 Processing Systems: A Survey.* Springer, 2016.
[30] W. Thies et al. StreamIt: A compiler for streaming applications. Technical Report LCS-TM-622, MIT, 2002.
[31] A. Toshniwal et al. Storm @Twitter. In *SIGMOD*, 2014.
[32] T. T. Tran et al. PODS: A new model and processing algorithms for uncertain data streams. In *SIGMOD*, 2010.
[33] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.
[34] W. Zhang, M. Hirzel, and D. Grove. AQuA: Adaptive quality analytics. In *DEBS*, 2016.