Beyond Analytics: the Evolution of Stream Processing Systems

Paris Carbone
paris.carbone@ri.se
RISE

Marios Fragkoulis
m.fragkoulis@tudelft.nl
Delft University of Technology

Vasiliki Kalavri
vkalavri@bu.edu
Boston University

Asterios Katsifodimos
a.katsifodimos@tudelft.nl
Delft University of Technology

ABSTRACT

Stream processing has been an active research field for more than 20 years, but it is now witnessing its prime time due to recent successful efforts by the research community and numerous worldwide open-source communities. The goal of this tutorial is threefold. First, we aim to review and highlight noteworthy past research findings, which were largely ignored until very recently. Second, we intend to underline the differences between early ('00-'10) and modern ('11-'18) streaming systems, and how those systems have evolved through the years. Most importantly, we wish to turn the attention of the database community to recent trends: streaming systems are no longer used only for classic stream processing workloads, namely window aggregates and joins. Instead, modern streaming systems are being increasingly used to deploy general event-driven applications in a scalable fashion, challenging the design decisions, architecture and intended use of existing stream processing systems.

ACM Reference Format:
Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: the Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3318464.3383131

1 INTRODUCTION

During the last 10 years, applications of stream processing technology have gone through a resurgence, penetrating multiple and very diverse industries. Nowadays, virtually all Cloud vendors offer first-class support for deploying managed stream processing pipelines, while streaming systems are used in a variety of use-cases that go beyond the classic streaming analytics (windows, aggregates, joins, etc.). For instance, web companies are using stream processing for dynamic car-trip pricing, banks apply it for credit card fraud detection, while traditional industries apply streaming technology for real-time harvesting analytics.

As seen in Figure 1, during the last 20 years, streaming technology has evolved significantly, under the influence of database and distributed systems. The notion of streaming queries was first introduced in 1992 by the Tapestry system [47], and was followed by lots of research on stream processing in the early 00s. Fundamental concepts and ideas originated in the database community and were implemented in prototypical systems such as TelegraphCQ [20], Stanford's STREAM, NiagaraCQ [21], Auroral/Borealis [1], and Gigascope [22]. Although these prototypes roughly agreed on the data model, they differed considerably on querying semantics [5, 13]. This research period also introduced various systems challenges such as sliding window aggregation [6, 36], fault-tolerance and high-availability [8, 44], as well as load balancing and shedding [46]. This first wave of research was highly influential to commercial stream processing systems that were developed in the following years (roughly during 2004 – 2010), such as IBM System S, Esper, Oracle CQL/CEP and TIBCO. These systems focused – for the most part – on streaming window queries and Complex Event Processing (CEP). This era of systems was mainly characterized by scale-up architectures processing ordered event streams.

The last reincarnation of streaming systems has come as a result of stream processing research that started roughly after the introduction of MapReduce [23] and the popularization of Cloud Computing. The focus shifted towards distributed,

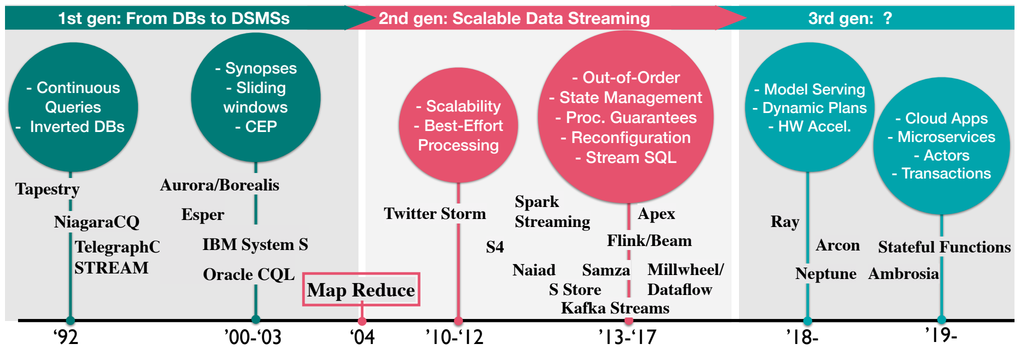

Figure 1: An overview of the evolution of stream processing and respective domains of focus.

data-parallel processing engines and shared-nothing architectures on commodity hardware. Lacking well-defined semantics and a proper query language, systems like Millwheel [3], Storm, Spark Streaming [50], and Apache Flink [16] first exposed primitives for expressing streaming computations as hard-coded dataflow graphs and transparently handled data-parallel execution on distributed clusters. With very high influence, the Google Dataflow model [4] re-introduced older ideas such as out-of-order processing [37] and punctuations [49] proposing a unified parallel processing model for streaming and batch computations. Stream processors of this era are converging towards fault-tolerant, scale-out processing of massive out-of-order streams.

At the moment of writing we are witnessing a trend towards using stream processors to build more general event-driven architectures [34], large-scale continuous ETL and analytics, and even microservices [33]. These use-cases have led to application designs where the stream processor's *state* has become a first-class citizen, visible to programmers [15] but also to external systems.

We believe that this is the right moment to reflect on the similarities and differences between the two eras of streaming research and to assemble lessons learned. Moreover, we wish for this tutorial to bring forward important, but overlooked works that have played a vital role in today's streaming systems design. Finally, we aim to establish a common nomenclature for concepts which various open-source projects and commercial offerings are misusing.

1.1 Tutorial Outline

We will split this tutorial in three parts.

1. Review of stream processing foundations. This introductory part covers fundamentals of stream processing computation and system design. Specifically, we will cover languages and semantics, notions of time, out-of-order handling, and progress mechanisms.

2. Evolution of system aspects. The second part discusses the evolution of streaming system aspects. In particular, we will review state management practices, fault recovery, high availability and load management techniques.

3. Emerging applications & new requirements. In the third part, we describe the characteristics and requirements of emerging streaming applications, focusing on real-time training and serving of ML models, event-driven applications, and microservice pipelines. We then propose future research directions.

1.2 Practical information

Scope. This survey tutorial provides an overview of the two eras of research in stream processing, highlighting impactful work that has shaped modern streaming systems. We then discuss emerging applications, new requirements, and challenges to be addressed by the research community.

Prior tutorials. The material proposed in this tutorial has not been presented before. Prior tutorials have either focused on a specific area (e.g., streaming optimizations [43]) or on the broader area of event processing [24]. To the best of our knowledge, the tutorial closest to ours was presented at DEBS 2014 [28] and covered Cloud Computing aspects of stream processing. Since then, modern streaming systems and applications have matured enough to allow for a deeper retrospective analysis and comparison.

Duration & target audience. The tutorial will last three (3) hours. Our target audience is graduate students, researchers, and practitioners from industry, who would benefit from a broader perspective of the evolution of stream processing. No prior knowledge of stream processing concepts is required as a prerequisite for attendance.

2 REVIEW OF FOUNDATIONS

The first part of our tutorial will be used to establish a common understanding of the basic notions of query languages, the effect of event ordering, as well as the time dimensions of streams. Finally, we will present various definitions of processing-progress in (dis)ordered streams.

2.1 Languages and Semantics

Streaming query languages have been a subject of research since the very first days of stream processing. Virtually every attempt to create a standard language for streams has been an extension of SQL, by adding windowing and means to convert from streams to relations and vice versa. Most noteworthy examples were CQL [5] and its derivatives [10, 31]. Later, dozens of works tried to extend the same standard for niche use-cases, with custom window types and aggregates; none of those attempts made it to standards.

Under the influence of MapReduce-like APIs, the majority of open-source streaming systems implemented functional/fluent APIs embedded in general purpose programming languages to hardcode Aurora-like dataflows. Till today, various communities are working on establishing a language for expressing computations which combine streams and relational tables, without a clear consensus.

2.2 Time and Order

Time and order lie at the heart of unbounded data processing. Due to stochastic factors like network latency and operations like shuffling and partitioning, data often arrives at a streaming system out of order. Besides the causes and effects of out-of-orderness, the tutorial will examine the two fundamental strategies for processing out-of-order data. The first buffers data at the ingestion point and allows batches of data to proceed in order [3, 37, 45, 49]. The second, ingests out-of-order data as they arrive and adjusts the computations in face of late data [9, 41].

2.3 Tracking Processing Progress

Streaming systems need a way to track processing progress, i.e., how far along a stream has been processed. Progress tracking is required for triggers, windowing and state purging. Multiple measures have been devised to track progress. These measures are: punctuations [49], watermarks [4], heartbeats [45], slack [1], and frontiers [40]. In the tutorial we will compare and contrast these mechanisms with examples.

3 EVOLUTION OF SYSTEM ASPECTS

Although the foundations of stream processing have remained largely unchanged over the years, important system aspects have transformed streaming systems into sophisticated and scalable engines, producing correct results in the presence of failures.

3.1 State Management

State is a concept that has always been central to stream processing. The notion of state itself has been addressed with many names in the past, such as "summary", "synopsis", "sketch" or "partitioned state", and that reflects aspects of the evolution of data stream management over time. For example, several early systems adopted a bounded memory model for predefined stream operations such as window aggregates and joins with actual state being a best-effort, approximate summarization of necessary stream statistics for the operation at hand. In other cases, underlying stream runtimes had been oblivious of data structures and variables defined within the user scope of a stream application, leaving every challenge related to state management to the programmer.

The need for explicit state management originates from the need to keep and automate the maintenance of persistent state for event-driven applications in a reliable manner. That includes the ability to store state beyond main memory, offer transactional processing guarantees, and allow for system reconfiguration [15, 17, 29]. Such requirements made it necessary to design systems fully aware of stream state and capable of managing all associated operations transparently.

During our tutorial we are going to spend a considerable amount of time explaining the challenges related to partitioned state and processing guarantees, highlighting their impact to system design. We will further classify approaches into two broad directions: (i) internally- [15, 17, 42] and (ii) externally-managed state [3, 18, 38], encapsulating whether state is managed within or outside a stream processor. We will discuss related aspects and technologies such as file systems, log-structured merge trees and related data structures, state expiration policies, window state maintenance, checkpoints, lineage-based approaches [50] and partitioning schemes for maintaining state.

3.2 Fault Recovery and High Availability

Fault recovery and high availability (HA) have been primary concerns for stream processing systems [30]. Apart from maintaining guarantees upon partial failures, stream processors also aim for low-latency executions. Therefore, recovery and HA mechanisms have to be non-obstructive to the stream execution at hand.

We will review the evolution of two common HA techniques: *active* and *passive* standby. An *active* standby runs two identical processing task instances in parallel and switches to the secondary instance when a fault occurs in the primary. This approach secures the highest level of availability and is the preferred option for critical applications. In contrast, a *passive standby* instantiates a new instance of a failed operator on idle resources, such as a provisioned virtual machine [15, 17]. Passive standby has gained traction recently with the scale-out capacity of streaming systems. The modern version of passive standby entails transferring the computation code and the latest checkpointed state snapshot of a failed operator instance to an available compute node, such as a virtual machine or a container, and resuming operation from the latest checkpoint.

3.3 Elasticity and Load Management

Due to the push-based nature of streaming inputs from external data sources, input rates might exceed system capacity causing performance degradation and increased latency. To counter this situation stream processing systems employ load management techniques. This is an area where the old and new systems contrast very vividly.

Early on, systems used *load shedding* techniques to address excess traffic by dynamically dropping tuples. The load shedder system is expected to be capable of deciding when, where (in the query plan), how many, and which tuples to drop so that (i) latency improves to an acceptable level and (ii) results quality degrades only minimally. In contrast, modern stream processing systems rely on managed partitioned state and the resource abundance of the Cloud and respond to workload variation via *elasticity* [17, 26, 32]. Elastic stream processors continuously monitor application performance and perform scale-out or scale-in actions of individual operators, ensuring correct migration of state partitions. In situations where input sources can control the data production rate, streaming systems can also utilize backpressure to inform the input sources to slow down.

4 PROSPECTS

Since its very beginning, stream processing has been conceived as a means of querying unbounded data sources in a relational fashion. The early systems and languages were designed as extensions of relational execution engines, with the addition of windows. As reflected by the current commercial stream processor offerings, traditional applications have been, for the most part, streaming analytical queries and CEP. Current streaming systems have evolved in the way they reason about completeness and ordering (e.g., out-of-order computation) and have witnessed architectural paradigm shifts that constituted the foundations of processing guarantees, reconfiguration, and state management. In this part of the tutorial, we summarize such observations from both application and system emerging needs. We then discuss how these new requirements can shape key characteristics of the future generation of data stream technology and outline open problems towards that direction.

4.1 Emerging Applications

Cloud Applications. At the moment of writing, we observe an interesting split in the design of modern Cloud programming frameworks. On one side of the spectrum, we witness stream processing APIs being implemented on top of Actor-like programming models – examples of this is Orleans [11, 14] and Akka Streams, as well as the streaming API of the Ray project [39]. On the other side of the spectrum we see stream processing technology being used as a backend for Actor-like abstractions such as Stateful Functions[1] [2] tailored for Cloud deployment. These two paradigms (Actors on streams vs. streams on Actors) signify that the streaming and Actor programming paradigms and their associated system implementations might be converging, as a result of technology maturation and application requirements. We believe that stream processors can become full-fledged systems for backing Cloud services such as Virtual Actors [11] and Microservices [27, 33], capable of executing transactions, performing analytics, and embedding complex business logic of stateful services inside dataflow operators.

Machine Learning. At the moment of writing, ML models are typically trained offline and a stream processor is used for model serving. Alternatively, the stream processor runtime is used for data distribution and coordination but complex operations, such as training and inference, are still mainly performed by specialized libraries. Thus, operators need to issue RPC calls to external ML frameworks and model servers adding both latency and complexity to the pipeline. Moreover, ML models need to be continuously updated, and oftentimes trained within the same pipeline as model serving. This means that the stream processor can cover the needs for online training, by offering constructs such as iterations, dynamic tasks, and shared state.

Streaming Graphs. Another emerging application area is continuous analysis of graph streams, where events indicate edge and vertex additions, deletions, and modifications. Even though there exist many specialized systems for dynamic graph processing [12], modern stream processors do not inherently support graph streaming use-cases. A prominent use-case is traffic and demand prediction for ride sharing services. Such an application needs to continuously compute shortest path queries with low latency and solve challenging online graph learning problems at the same time. It needs to simultaneously learn from, and analyze evolving graph inputs and other structured or unstructured types of static and dynamic data, such as driver and user locations, rush hours per area, neighborhood and points of interest (theaters, stadiums, etc.). The prediction tasks require generating graph embeddings using streaming random walks or online neural network training. Another use-case is online network management in SDN controllers, where real-time events update the network topology and controllers execute continuous routing decisions, evaluate verification tasks, and find backup paths for each link in a streaming fashion.

4.2 The road ahead

Several aspects of streaming systems could evolve further by exploiting next-generation hardware and incorporating ideas

[1]http://statefun.io

	Programming Models	Transactions	Advanced State Backends	Loops & Cycles	Elasticity & Reconfiguration	Dynamic Topologies	Shared Mutable State	Queryable State	State Versioning	Hardware Acceleration
Cloud Apps	✓	✓	✓	✓	✓	✓		✓	✓	
Machine Learning	✓		✓	✓		✓	✓	✓	✓	✓
Graph Processing	✓		✓	✓			✓			

Table 1: Requirements for new applications

from programming languages, domain specific models, and distributed computing. During the tutorial we will discuss the following requirements that stem from new applications.

Programming Models. Modern streaming systems allow developers to author streaming topologies with user-defined functions and functional APIs [7, 16] or some variant of streaming SQL [10]. However, these make the development of event-driven, Cloud applications very cumbersome. In fact, developers can only develop Cloud apps in a very low-level dataflow API. To build loosely-coupled Cloud apps, we need novel APIs that will allow developers to author simple high-level functions [2] or actor-like APIs [14, 39] that can be compiled into streaming dataflows. Machine Learning and graph processing workloads require programming models and abstractions that allow for iterations and complex data types (e.g., matrices, graphs), instead of tuple-like events.

Transactions. Streaming systems lack transactional facilities to support advanced business logic and coordination as required by Cloud applications with the exception of S-Store [38], which provides ACID guarantees on shared mutable state on a single machine. Cloud applications feature many use-cases that span across components/services in a typically distributed environment. Coordination of the components' processing is essential to safeguard state consistency and provide a single success or fail response that mirrors the successful recording of all state changes or none at all. Applications require support by programming frameworks for expressing transaction workflows that involve multiple components and for handling transaction abort cases and rollback actions in an automated manner.

Advanced State Backends. A popular choice for a state backend in streaming systems is a key-value store [15, 19]. However, Cloud apps as well as machine learning and graph processing applications may require more sophisticated state such as dense matrices, large relational tables or an object-based blob storage. Despite the chosen data model, persistently-backed state may need to be indexed and cached for fast access (think of product images qualified by a product id), support transactions including the possibility of a rollback (e.g., for online payments), or process complex analytical queries (e.g. on top of a Cloud data warehouse). Applications need a way to provide system-wide guarantees across the potentially disparate backends that comprise its overall state such as fault tolerance with strict consistency.

Loops & Cycles. Most dataflow systems today limit computation primarily in DAGs due to limitations in flow control (deadlock elimination) and monotonic event-time progress estimation. Nevertheless, there is a critical need for loops, either in the form of asynchronous event feedback or synchronously using bulk iterative semantics (Bulk and Stale Synchronous model variants). Asynchronous loops can enable request and response logic to support Cloud apps that demand two-way message exchange across functions (e.g., serverless) or actors or other more sophisticated mechanisms on dataflow tasks such as concurrency control for transaction lifecycles. Synchronous loops are paramount for bulk iterative algorithms used in machine learning (e.g., Stochastic Gradient Descent) and are also critical for graph analytics that rely on iterative superstep synchronization to ensure consistent results. Therefore, the ability to express and execute different forms of loops will enable in the future a large set of computational models to be subsumed by stream processors. Despite existing efforts such as the Timely Dataflow model implemented in Naiad [40], there is still a need for intuitive and compositional integration of loops in existing systems at the programming model to allow users to express iterative operations while inter-playing seamlessly with event-time out-of-order processing [37].

Elasticity & Reconfiguration. Streaming systems provide limited means for elasticity and reconfiguration actions, such as changing resource allocations and updating operator logic amidst a job execution. Typically a stream processing job has to save its state, terminate its execution, and restart it with the refreshed operators. For Cloud applications that have to be constantly online, this support is inadequate. Instead, applications need to apply code updates and hot fixes seamlessly to their operation without affecting the state or the processing of user requests.

Dynamic Topologies. The conventional means of expressing and executing dataflow streaming applications as statically compiled and scheduled graphs has been a limiting factor in both expressibility and performance for several types of computations. Many Cloud apps are dynamic by nature, requiring new instances of service components to be spawned on demand and execute their event-based logic independently from the "main" dataflow. Likewise, ML pipelines such as those seen in reinforcement learning are built around the notion of expanding computation dynamically during model exploration [39]. The ability to compose dataflow topologies dynamically beyond static stream tasks can aid such application domains and also offer new performance capabilities to existing streaming use-cases such as work-stealing, parallel-recovery, skew mitigation and parallel execution of global aggregates (e.g., global windows) on demand.

Shared Mutable State. A large set of applications from computational fields such as simulations, ML task-driven model training, and graph aggregations rely on the availability of shared mutable state, i.e., shared variables where multiple tasks can read and write. While this approach appears unconventional and challenging to integrate in purely-data parallel streaming systems, it carries great potential towards supporting non key-parallel modes of operation, model optimization algorithms and more complex data types.

Queryable State. Stream processing applications build and enrich persistent large state that represents dynamic relational tables, ML feature matrices or other types of derivative results from pre-processed and joined data from multiple input streams. While the standard interaction point with a stream processor has always been its input and output streams, internal state, currently a black box to the user, is becoming the main point of interest for many interactive and reactive data applications today. A step towards better reuse of computation is to allow dataflow applications to subscribe and gain read access to intermediate views of their respective states. This capability can further boost better interoperability across different Cloud apps and their internal components (e.g., stateful functions) as well as training and serving logic in ML. Related challenges in that direction include external query access isolation (e.g., snapshot) and flexible state management, which have only been partially considered in existing solutions (S-Store [38], Flink point-queries [15]).

State Versioning. Streaming systems provide no explicit support for state versioning and state schema evolution. For Cloud and machine learning applications however, mutating the state schema is part of their lifecycle. Cloud applications change frequently, e.g., to introduce new services or update the current ones. As their state schema evolves, applications need a reliable way to version their state in order to continue operating consistently. Machine Learning applications also require state versioning. For instance, consider a continuous model serving pipeline (e.g., fraud detection) where a ML model needs to be updated while the pipeline is running.

Hardware Acceleration. Hardware accelerators such as GPUs, TPUs, and FPGAs have become mainstream for certain ML workloads, especially when tensor computation is involved. While hardware acceleration has never really been a hard requirement for data streaming it starts to become more relevant, given the broadening capabilities of stream processors (e.g., Stream ML). Recent findings [35, 51] have shown that stream-native operations (e.g., window aggregation) can also benefit from hardware accelerators such GPUs and Cloud FPGAs [48]. Overall, there is potential for more specialized code generation targeting diverse hardware architectures and general-purpose stream operators in the future [51]. Furthermore, apart from speedups, modern hardware can also lead to new capabilities in stream processing that were not previously considered possible. For example, new storage and network hardware can enable novel fault tolerance and state management mechanisms. Managed state currently resides mostly in *volatile* memory and can be lost upon failure. The potential adoption of NVRAM and RDMA within compute clusters could shift current approaches from fail-stop to efficient fault-recovery models [17, 25].

5 PRESENTER BIOGRAPHIES

Paris Carbone is a senior systems researcher at RISE, serving as the leader of the Continuous Deep Analytics group. Paris holds a PhD in distributed computing from KTH and is one of the core committers for Apache Flink® with key contributions to its state management.

Marios Fragkoulis is a postdoctoral researcher at TU Delft, working on scalable stream processing. Marios holds a PhD in main memory data analytics from the Athens University of Economics and Business. He is the co-creator of dgsh, the directed graph shell.

Vasiliki Kalavri is an Assistant Professor at Boston University Department of Computer Science. She is working on distributed stream processing and large-scale graph analytics and has a joint PhD from KTH and UCLouvain. Vasiliki is a PMC member of Apache Flink® and a co-author of the *Stream Processing with Apache Flink* book.

Asterios Katsifodimos is an Assistant Professor at TU Delft. His research focuses on scalable data management and stream processing, and received the ACM SIGMOD Research Highlights award in 2016 and the Best Paper Award in EDBT 2019. Before TU Delft, Asterios worked at SAP and TU Berlin, and received his PhD from INRIA in Paris.

6 ACKNOWLEDGEMENTS

This work is partly funded by the Swedish Foundation for Strategic Research under Grant No.: BD15-0006, the EU H2020 OpertusMundi project No.870228 and Delft Data Science.

REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDBJ*, 2003.

[2] A. Akhter, M. Fragkoulis, and A. Katsifodimos. Stateful functions as a service in action. Demo. *In VLDB*, 2019.

[3] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, 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*, 2015.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDBJ*, 2006.

[6] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[7] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *SIGMOD*, 2018.

[8] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM TODS*, 2008.

[9] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *CoRR*, abs/cs/0612115, 2006.

[10] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *SIGMOD*, 2019.

[11] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[12] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of Streaming and Dynamic Graphs: Concepts, Models, Systems, and Parallelism. *CoRR*, abs/1912.12740, 2020.

[13] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *In VLDB*, 2010.

[14] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing*, 2011.

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *In VLDB*, 2017.

[16] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[18] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik. S-Store: A streaming NewSQL system for big velocity applications. *In VLDB*, 2014.

[19] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. In *SIGMOD*, 2018.

[20] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

[21] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *SIGMOD Record*, 2000.

[22] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008.

[24] O. Etzion. Event processing: Past, present and future. *In VLDB*, 2010.

[25] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, 2014.

[26] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in Heron. *In VLDB*, 2017.

[27] J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, R. G. Peshawaria, T. Zaccai, and I. Zhang. A.M.B.R.O.S.I.A: Providing performant virtual resiliency for distributed applications. *In VLDB*, page 588–601, Jan. 2020.

[28] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak. Cloud-based data stream processing. In *DEBS*, 2014.

[29] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *In VLDB*, 2019.

[30] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[31] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. *In VLDB*, pages 1379–1390, 2008.

[32] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*. USENIX Association, 2018.

[33] A. Katsifodimos and M. Fragkoulis. Operational stream processing: Towards scalable and consistent event-driven applications. In *EDBT*, 2019.

[34] M. Kleppmann, A. R. Beresford, and B. Svingen. Online event processing: Achieving consistency where distributed transactions have failed. *ACM Queue*, 2019.

[35] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*. ACM, 2016.

[36] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 2005.

[37] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *In VLDB*, 2008.

[38] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-Store: Streaming meets transaction processing. *In VLDB*, 2015.

[39] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging AI applications. In *OSDI*, 2018.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[41] C. Mutschler and M. Philippsen. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *DEBS*, New York, NY, USA, 2013. ACM.

[42] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at Linkedin. *In VLDB*, 2017.

[43] S. Schneider, M. Hirzel, and B. Gedik. Tutorial: Stream processing optimizations. In *DEBS*. ACM, 2013.

[44] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[45] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*. ACM, 2004.

[46] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320. VLDB Endowment, 2003.

[47] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD Record*, 1992.

[48] J. Thomas, P. Hanrahan, and M. Zaharia. Fleet: A framework for massively parallel streaming on FPGAs. In *ASPLOS*, 2020.

[49] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 2003.

[50] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *USENIX HotCloud*, 2012.

[51] S. Zhang, F. Zhang, Y. Wu, B. He, and P. Johns. Hardware-conscious stream processing: A survey. *SIGMOD Record*, 2020.