Towards a Unified Ingestion-and-Storage Architecture for Stream Processing

Ovidiu-Cristian Marcu*, Alexandru Costan†, Gabriel Antoniu*, María S. Pérez-Hernández‡, Radu Tudoran§, Stefano Bortoli§ and Bogdan Nicolae§
*Inria Rennes - Bretagne Atlantique, France
†IRISA / INSA Rennes, France
‡Universidad Politecnica de Madrid, Spain
§Huawei Germany Research Center

***Abstract*—Big Data applications are rapidly moving from a batch-oriented execution model to a streaming execution model in order to extract value from the data in real-time. However, processing live data alone is often not enough: in many cases, such applications need to combine the live data with previously archived data to increase the quality of the extracted insights. Current streaming-oriented runtimes and middlewares are not flexible enough to deal with this trend, as they address ingestion (collection and pre-processing of data streams) and persistent storage (archival of intermediate results) using separate services. This separation often leads to I/O redundancy (e.g., write data twice to disk or transfer data twice over the network) and interference (e.g., I/O bottlenecks when collecting data streams and writing archival data simultaneously). In this position paper, we argue for a unified ingestion and storage architecture for streaming data that addresses the aforementioned challenge. We identify a set of constraints and benefits for such a unified model, while highlighting the important architectural aspects required to implement it in real life. Based on these aspects, we briefly sketch our plan for future work that develops the position defended in this paper.**

***Index Terms*—Big Data, Streaming, Storage, Ingestion, Unified Architecture**

I. Introduction

Under the pressure of increasing data velocity, Big Data analytics shifts towards stream computing: it needs to deliver insights and results as soon as possible, with minimal latency and high frequency that keeps up with the rate of new data. In this context, online and interactive Big Data runtimes designed for stream computing are rapidly evolving to complement traditional, batch-oriented runtimes (such as MapReduce [1]) that are insufficient to meet the need for low latency and high update frequency [2].

This online dimension [3], [4] of data processing was recognized in both industry and academia and led to the design and development of an entire family of online Big Data analytics runtimes: Apache Flink [5], Apache Spark Streaming [6], Streamscope [7], Apache Apex [8], etc.

As depicted in Figure 1, a typical state-of-art online big data analytics runtime is built on top of a three layer stack:

- **Ingestion**: this layer serves to acquire, buffer and optionally pre-process data streams (e.g., filter) before they are consumed by the analytics application. The ingestion layer does not guarantee persistence: it buffers the data only temporarily and enables limited access semantics to it (e.g., it assumes a producer-consumer pattern that is not optimized for random access)
- **Storage**: unlike the ingestion layer, this layer is responsible for persistent storage of data. This typically involves either the archival of the buffered data streams from the ingestion layer or the storage of the intermediate stream analytics results, both of which are crucial to enable fault tolerance or deeper, batch-oriented analytics that complement the online analytics.
- **Processing**: this layer consumes the streaming data buffered by the ingestion layer and sends the output and intermediate results to the storage or ingestion layers.

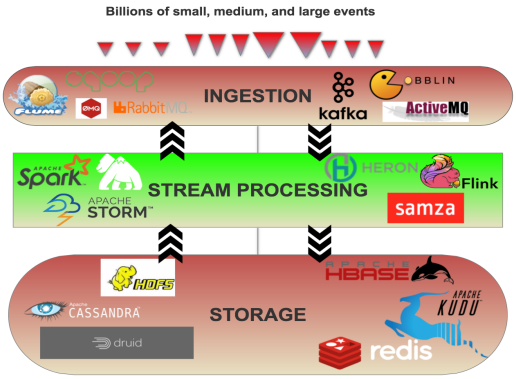

Fig. 1: The usual streaming architecture: data is first ingested and then it flows through the processing layer which relies on the storage layer for storing aggregated data or for archiving streams for later usage.

In practice, each layer is implemented as a dedicated solution that is independent from the other layers, under the assumption that specialization for a particular role enables better optimization opportunities. Such an architecture is commonly referred to as the *lambda* [9] architecture.

However, as the need for more complex online data manipulations arises, so does the need to enable better coupling between these three layers. Emerging scenarios (Section IV) emphasize complex data pre-processing, tight fault tolerance and archival of streaming data for deeper analytics based on batch processing. Under these circumstances, data is often written twice to disk or send twice over the network (e.g.

as part of a fault tolerance strategy of the ingestion layer and the persistency requirement of the storage layer). Second, the lack of coordination between the layers can lead to I/O interference (e.g. the ingestion layer and the storage layer compete for the same I/O resources). Third, the processing layer often implements custom advanced data management (e.g. operator state persistence, checkpoint-restart) on top of inappropriate basic ingestion/storage API, which results in significant performance overhead.

In this paper, we argue that the aforementioned challenges are significant enough to offset the benefits of specializing each layer independently of the other layers. To this end, we discuss the potential benefits of a *unified storage and ingestion architecture*. Specifically, we contribute with a study on the general characteristics of data streams and the requirements for the ingestion and storage layer (Section II), an overview of state-of-art and its limitations and missing features (Section III) and a proposal for a unified solution based on a series of design principles and an architecture (Section V).

II. Requirements of Ingestion and Storage for Data Streams

In this section we focus on the characteristics of data streams and discuss the main requirements and constraints of the ingestion and storage layers.

A. Characteristics of Data Streams

Data Size. Stream data takes various forms and most systems do not target a certain size of the stream event, although this is an important characteristic for performance optimizations [10]. However, a few of the analyzed systems are sensitive to the data size, with records defined by the message length (from tens or hundreds of bytes up to a few megabytes).

Data Access Pattern. Stream-based applications dictate the way data is accessed, with most of them (e.g., [3]) requiring fine-grained access. Stream operators may leverage multi-key-value interfaces, in order to optimize the number of accesses to the data store or may access data items sequentially (scan based), randomly or even clustered (queries). When coupled with offline analytics, the data access pattern plays a crucial role in the scalability of wide transformations [11].

Data Model. This describes the representation and organization of single information items flowing from sources to sinks [12]. Streams are modeled as simple records (optional key, blob value for attributes, optional timestamp) or row/column sets in a table. Data model together with data size influence the data rate (i.e., throughput), an important performance metric related to stream processing.

Data Compression. In order to improve the consumer throughput (e.g., due to scarce network bandwidth), or to reduce storage size, a set of compression techniques (e.g., GZIP, Snappy, LZ4, data encodings, serialization) can be implemented and supported by stream systems.

Data Layout. The way data is stored or accessed influences the representation of stream data in memory or on disk. The need to query (archived) stream data (reads or updates) forces the data layout to be represented in a columnar oriented approach [13] or a hybrid row-column layout may be necessary.

B. Functional Requirements

Adequate Support for Data Stream Partitioning. A stream partition is a (possibly) ordered sequence of records that represent a unit of a data stream. Partitioning for streaming is a recognized technique used in order to increase processing throughput and scalability, e.g., [14], [15]. We differentiate between data partitioning techniques implemented by ingestion/storage and application-level (user-defined) partitioning [16], [17], as it is not always straightforward to reconcile these two aspects.

Support for Versatile Stream Metadata. Streaming metadata is small data (e.g., record's key or attributes) that describes stream data and it is used to easily find particular instances of data (e.g., to index data by certain stream attributes that are later used in a query).

Support for Search. The search capability is a big challenge: this is more appropriate to specialized storage systems. Ingestion frameworks usually do not support such feature. A large number of stream applications [18] need APIs able to scan (random or sequential) the stream datasets and support to execute real-time queries.

Advanced Support for Message Routing. Routing defines the flow of a message (record) through a system in order to ensure it is processed and eventually stored. Applications may need to define necessary dynamic routing schemes or can rely on predefined static routing allocations. Readers can refer to [19] for a characterization of message transport and routing.

Backpressure Support. Backpressure refers to the situation where a system cannot sustain any more the rate at which the stream data is received. It is advisable for streaming systems to durably buffer data in case of temporary load spikes, and provide a mechanism to later replay this data (e.g., by offsets in Kafka). For example, one approach for handling fast data stream arrivals is by artificially restricting the rate at which tuples are delivered to the system (i.e., rate throttling), this technique could be implemented either by data stream sources or by stream processing engines.

C. Non-functional Requirements

High Availability. Failures can disrupt or even halt stream processing components, can cause the loss of potentially large amounts of stream processed data or prevent downstream nodes to further progress: stream processing systems should incorporate high-availability mechanisms that allow to operate continuously for a long time in spite of failures [20], [21].

Temporal Persistence versus Stream Durability. Temporal persistence refers to the ability of a system to temporarily store a stream of events in memory or on disk (e.g., configurable data retention period of a short period of time after which data is automatically discarded). For example, windowing [22] is an important stream primitive and its main characteristic is the window's state (i.e., a set of recent tuples) that needs to be temporarily persisted. Stream records may

Requirement	Kafka	Redis	Hyperion	Druid	Kudu
Size	Configurable message length	Up to 512 MB. Optimized data types	KBs	Dimension and metric columns (KBs, MBs)	Typed columns (up to 64KB)
Access	Record, logical offsets	Fine-grained; multi-group queries	Queries	Queries (search, time-series)	Fine-grained, scans, queries (time-series)
Model	Records as key, payload, opt. timestamp	Key-Value, data structures	Immutable stream records	Data tables (time-stamped events)	Tables
Compression	Relative offsets; LZ4, GZIP, Snappy; log compaction	Data encodings	Non-goal	Dictionary encoding for strings, LZF for numeric	Pattern-based on typed columns, LZ4, Snappy, zlib
Layout	On disk logs of ordered, immutable sequence of records	In-memory backed by disk (custom checkpoints)	On-disk log-structured, fixed blocks of 1MB	Segments stored as columns	Columnar (memory and disk row sets)

TABLE I: Data characteristics

also need to be durably stored: it is of utmost importance to be able to configure a diverse set of retention policies in order to give the ability to replay historical stream events or to derive later insights by running continuous queries over both old and new stream data.

Scalability. Scalability is the ability of the stream storage to handle increasing amounts of data, while remaining stable in front of peaks moments when high rates of bursting data arrive. Scalable stream storage should scale either vertically (increase the number of a clients that are supported by a single-node broker/data node) or horizontally (distribute clients among a number of interconnected nodes).

Latency versus Throughput. Recognizing that it is not easy to offer both low latency and high throughput, especially in front of acknowledgment requirements imposed in some cases, a unified ingestion and storage system for streaming needs to be designed to allow for a trade-off between latency and throughput.

III. State-of-Art Overview and Limitations

In this section, we briefly introduce a series of state-of-art ingestion and storage systems for data streams and discuss their limitations.

A. Overview

Apache Kafka [23] is a distributed stream platform that provides durability and publish/subscribe functionality for data streams (making streaming data available to multiple consumers), being the de facto open-source solution (for temporal data persistence and availability) used in end-to-end pipelines with streaming engines.

Redis [24] is an in-memory data structure store that is used as a database, cache and message broker. Redis also implements the pub/sub messaging paradigm and groups messages into channels with subscribers expressing interest into one or more channels.

Hyperion [25] is a system for archiving, indexing, and online retrieval of high-volume packet header data streams. Data archiving for network monitoring systems is complicated by data arrival rates and the need for online indexing of this data to support retrospective queries.

Druid [26] is a distributed, columnar-oriented data store designed for real-time exploratory analytics on big data sets. Its motivation is straightforward: although Hadoop (HDFS) is a highly available system, its performance degrades under heavy concurrent load and is not optimized for ingesting data and making data immediately available for processing.

Apache Kudu [27] is a columnar data store that integrates with Apache Impala, HDFS and HBase. It can be seen as an alternative to Avro/Parquet over HDFS (not suitable for updating individual records or for efficient random reads/writes) or an alternative to semi-structured stores like HBase/Cassandra (not efficient for sequential read throughput).

B. Limitations

In Table I we comment on how ingestion and storage systems handle the characteristics of data streams introduced in Section II-A. Although these systems are designed to handle *records of different sizes*, it is not clear what optimizations are used to efficiently process them. *Compression* is another important goal, well supported by selected systems. When evaluating their stream use case, users should first assess the *data size* and *access characteristics* in order to understand whether the selected system is able to handle optimally such data. Users should make an evaluation study to validate the performance of the required access patterns when considering the stream data uncompressed and also with different compression methods.

In Table II we discuss how the ingestion and storage systems relate to the functional and non-functional requirements introduced in Section II-B and, respectively, Section II-C. We observe that some systems handle *data* as being *immutable* in order to optimize the write throughput and offer sometimes-specialized API for querying data. Storage systems implement different strategies for *data partitioning* (time-based, key-based, or custom solutions); *what is missing is the support for application-level partitioning* and it is not clear if these systems can easily adapt to particular user strategies for stream partitioning (e.g., partition by certain attribute). One important observation for performance aspects is that, although in some cases systems offer a static way of configuring a *trade-off between latency and throughput*, this important requirement remains a challenge. This could be better addressed by a unified ingestion and storage solution for streaming.

Requirement	Kafka	Redis	Hyperion	Druid	Kudu
Partitioning	Topics and partitions	Hash Slots (Cluster mode)	Streams partitioned into time intervals	Segments (shards data by time)	Range, hash, and multilevel partitioning
Metadata	Message header's attributes	Keys	Record and headers, block maps	Segments and configuration	Catalog tables, tablets
Search	Non-goal	Multi-get queries	Specific queries	JSON over HTTP, SQL	SQL
Routing	Kafka Connect (Timestamp and Regex Routers)	Key hashtag	Non-goal	IOConfig, Push/pull ingestion	Non-goal
Backpressure	Replay by consumer offsets; Quotas	Non-goal	Non-goal	Non-goal	Non-goal
Availability	Replicated partitions	Sentinel HA	Non-goal	Historical nodes	HA with Raft consensus
Scalability	Multi-Broker horizontal scalability	Multiple Server instances or Cluster mode	Multi monitors	Coordinator, broker, indexing, real-time nodes	Tablet servers and masters
Persistence	Temporal on-disk (retention policies)	Durable, in-memory backed on disk	Durable, on-disk	In memory and persisted indexes on real time nodes	Durable, on-disk
Latency	Depends on acknowledgements	Lower if used as cache	Dominated by per-record overhead	Depends on data dimensions	Efficient random access
Throughput	Batching size on producer	Pipelining multiple commands	Sequential, immutable writes	Throttle ingestion	Lazy materialization, delta compaction

TABLE II: Requirements for ingestion and storage systems.

C. Missing Features

Next, we identify a set of features that are currently missing or insufficiently addressed by state-of-art, yet crucial for the efficient processing of emerging online analytics scenarios.

Streaming SQL. Streaming queries or SQL on streams has recently emerged as a convenient abstraction to process flows of incoming data, inspired the deceptively simple decades-old SQL approach. However, they extend beyond time-based or tuple-based stream models [28]. Given the complexity of stream SQL semantics and the support they require for handling state for such operations, it is important to understand how a stream storage system can sustain and optimize such applications.

Access Pattern Detection. Ingestion/storage for streaming would both benefit from *detecting and then dynamically adapting* to the observed stream access patterns [29], [30], ranging from fine-grained per record/tuple access to group queries (multi get/put) or scan-based.

Windowing. A basic primitive in streaming is windowing [22] with its many forms (e.g., sliding, session). Streaming runtimes develop internal mechanisms to support the *window's state* (exactly once processing requires fault tolerant state). This complexity could be avoided at the processing level if the *required support for keeping the windowing state was developed within a unified ingestion/storage layer*.

User-Defined Aggregates. A popular technique to avoid moving large amounts of data over the network and avoid serialization and de-serialization overhead is to push data pre-processing routines close to where the data is stored, especially when considering modern storage equipped with processing capability. Also known as near-data processing, this feature would greatly benefit stream processing yet current ingestion/storage solutions do not offer native support for it.

IV. Use Cases Exhibiting Limitations and Missing Features

Stream processing can solve a large set of business and scientific problems, including network monitoring, real-time fraud detection, e-commerce, etc. In this section, we present two scenarios that exhibit the characteristics, requirements and missing features discussed above.

Monetizing streaming video content. This use case motivates the Dataflow [3] model proposed by Google for stream processing. Streaming video providers display video advertisements and are interested in efficiently billing their advertisers. Both video providers and advertisers need statistics about their videos (e.g., how long a video is watched and by which demographic groups); they need this information as fast as possible (i.e., in real-time) in order to optimize their strategy (e.g., adjust advertisers budgets). We identify a set of requirements associated to these applications: 1) events are *ingested* as fast as possible and consumed by processing engines that are updating statistics in real-time; 2) events and aggregated results are *stored* for future usage (e.g., offline experiments); 3) users *interrogate* streams (SQL queries on streams) to validate quality agreements.

Decision Support for Smart Cities applications. Future cities will leverage smart devices and sensors installed in the public infrastructure to improve the citizen's life. In this context, several aspects are important: 1) data from sensors may be initially *ingested and pre-processed before they are delivered* to the streaming engines; 2) massive quantities of data are received over short time intervals; 3) ingestion components have to support a high frequency of stream event rates.

Moreover, large web companies such as Twitter, LinkedIn, Facebook, Google, Alibaba need to ingest and log tens of millions of events per second. This trend is projected to grow

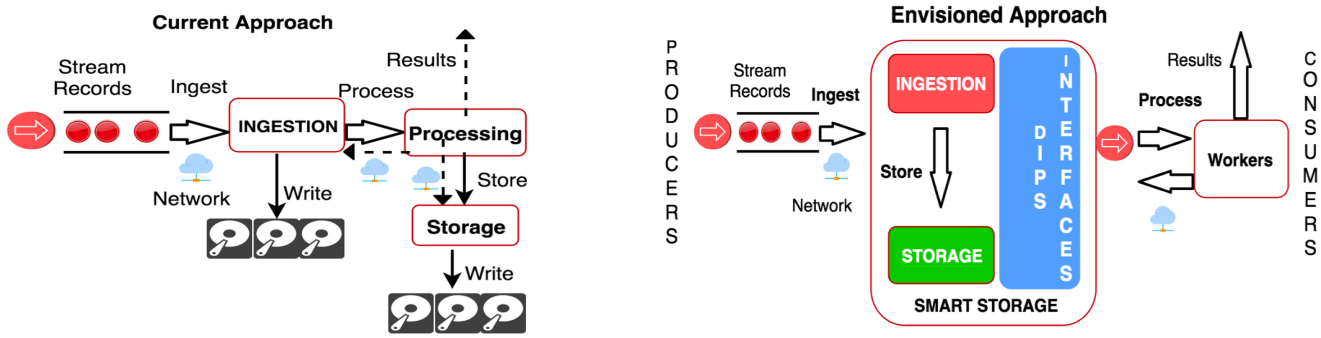

Fig. 2: Approaches for handling stream data ingestion and storage.

by 40% year over year (e.g., due to IoT, mobile games, etc.). However, only a small part of this data is ever accessed for processing (i.e., to extract value) after ingestion (less than 6%) [31]. Most of untouched data is usually archived (e.g., Facebook users are interested to maintain an archive of their actions and messages) and may be later queried. To sum up, stream-based applications strongly rely on the following features, not well supported by current streaming architectures:

1) *Fast ingestion*, doubled by *simultaneous indexing* (often, through a single pass on data) for real-time processing;
2) *Low-latency storage* with *fine-grained query support* for efficient filtering and aggregation of data records;
3) Storage coping with events accumulating in *large volumes over a short period of time*.

V. A Unified Stream Ingestion and Storage Proposal

In this section, we introduce our proposal for a unified stream ingestion and storage layer that is capable to address the limitations of the corresponding separated state-of-art layers and the missing features discussed in Section III. First, we introduce a set of design principles for stream processing that guide the design of a unified ingestion and storage layer. Then, we present our unified proposal that can handle unbounded streams while cooperating closer with processing engines, e.g., sharing the stream partitioning strategy with processing engines through DIPS interfaces and allowing *pushing processing functions next to stream data*. We think that current hardware trends (i.e., multi-core nodes, near data processing enabled storage devices, etc.) encourage the implementation of such unified model, allowing co-location of processing and data management.

A. Design Principles

Stream processing should focus on stateful computations: how to transform data through a workflow composed of stateful operators. The main focus should be how to define the computation, when to trigger the computation and how to combine the computation with offline analytics.

Unified ingestion/storage should focus on high-level data management: both ingestion and storage are exposed through a common engine that is capable of leveraging synergies to avoid I/O redundancy and I/O interference arising when using independent solutions for the two aspects. This engine handles caching hierarchy, de-duplication, concurrency control, etc. Furthermore, all high-level data management currently implemented in the processing engine (fault tolerance, persistence of operator states, etc.) should be handled natively by the unified layer.

Common abstractions for the interactions between processing engines and storage systems should be designed. Storage systems and processing engines should understand DIPS interfaces (i.e., for data ingestion, processing, and storage for streams of records): they offer APIs to read, write, process and store streams of records.

B. Architecture

As illustrated in Figure 1, traditional Big Data streaming architectures typically span over three layers: ingestion, processing, storage. To better illustrate the I/O redundancy present in traditional approaches, note the life-cycle of data in Figure 2 (left hand side): each record is written twice to disk (once by the ingestion framework and then by the storage system) and it also traverses twice the network (for processing and storage).

Guided by the design principles presented in the previous section, we introduce a unified architecture illustrated in Figure 2 (right hand side). In this case, the network and disk overhead associated with I/O redundancy are dramatically reduced, as data is already replicated in the memory of the co-located storage and processing nodes, which can be leveraged when archiving streams. Moreover, stream records do not need to pass through the processing layer before they are sent for archiving (the *Store* action is handled internally by the unified layer based on additional stream metadata that flags stream records for archiving).

This unified proposal not only implements the high-level data management mentioned in the design principles, but also exposes the DIPS interfaces necessary for efficient cooperation between stream storage and processing engine:

1) The *Data Ingest* interface is leveraged by stream producers that write input streams but also by processing engine' workers that store processing state to local storage instances;

2) The *Data Store* interface is handled internally by the storage system, being used to permanently store streams when needed: this action can be done asynchronously based on stream metadata and hints sent by the processing engine;
3) The *Data Process* interface is bidirectionally exposed: first, it can be leveraged by processing engine to pull data from the stream storage; second, we envision future processing workflows sending process functions to storage whenever possible. avoid moving data but process it right to its source.

To optimally leverage this proposal, processing engines should be co-located with the storage nodes, leveraging multi-core (fat) nodes that are built with tens of cores and hundreds of GBs of DRAM. Moreover, we envision to build our proposal on top of robust low-level key-value stores that are capable of efficient fine-grain access (e.g., put/get, multi-write/multi-read) to the stream records.

Another interesting side-effect of our proposal is the capability to develop complex high-level stream-based workflows where *different* applications can easily communicate with each other by sharing a unified ingestion and storage layer. In this case, the benefit of avoiding I/O redundancy and interference is preserved across multiple application instances by simply sharing the unified layer among them.

VI. Conclusion

In this paper we discuss the main challenges (i.e, characteristics, requirements limitations and missing features) of data management for stream computing. Specifically, we identify not only limitations of state-of-art in form of limited control over I/O redundancy and interference due to separation of ingestion and storage into independent layers, but also a series of missing features that are not addressed by state-of-art but critical for modern online analytics applications. To this end, we propose a series of design principles and architecture towards a unified stream ingestion and storage solution that addresses these limitations and missing features. We are working towards the *Kera* prototype to demonstrate our position in future work.

Acknowledgment

This research was funded by Huawei HIRP OPEN and the BigStorage project (H2020-MSCA-ITN-2014-642963, under the Marie Sklodowska-Curie action).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on OSDI*. USENIX Association, 2004.
[2] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *ACM SIGMOD*, 2016, pp. 1087–1098.
[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, Aug. 2015.
[4] R. Tudoran, B. Nicolae, and G. Brasche, "Data multiverse: The uncertainty challenge of future big data analytics," in *IKC'16: 2nd International KEYSTONE Conference*, Cluj-Napoca, Romania, 2016, pp. 17–22.
[5] "Apache Flink." http://flink.apache.org.
[6] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, 2013, pp. 423–438.
[7] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: Continuous reliable distributed processing of big data streams," in *Usenix NSDI*, 2016, pp. 439–453.
[8] "Apache Apex." http://apex.apache.org/.
[9] M. John, A. Cansu, Z. Stan, T. Nesime, and D. Jiang, "Data ingestion for the connected world," in *CIDR, Online Proceedings*, 2017.
[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS*, 2012, pp. 53–64.
[11] B. Nicolae, C. Costa, C. Misale, K. Katrinis, and Y. Park, "Leveraging adaptive i/o to optimize collective data shuffling patterns for big data analytics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1663–1674, 2017.
[12] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.
[13] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *ACM SIGMOD*, 2008.
[14] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, Aug. 2014.
[15] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 23–32, Apr. 2013.
[16] B. Cagri and T. Nesime, "Scalable data partitioning techniques for parallel sliding window processing over data streams," in *8th Intl. Workshop on Data Mgmt. for Sensor Networks*, 2011.
[17] L. Cao and E. A. Rundensteiner, "High performance stream query processing with correlation-aware partitioning," *Proc. VLDB Endow.*, vol. 7, no. 4, pp. 265–276, Dec. 2013.
[18] S. Chandrasekaran and M. J. Franklin, "Streaming queries over streaming data," in *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002, pp. 203–214.
[19] C. Mitch, B. Hari, B. Magdalena, C. Donald, C. Ugur, X. Ying, and Z. Stan, "Scalable distributed stream processing," in *CIDR*, 2003.
[20] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *IEEE ICDE*, 2005, pp. 779–790.
[21] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *ACM SIGMOD*, 2004, pp. 827–838.
[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *ACM SIGMOD*, 2005, pp. 311–322.
[23] "Apache Kafka." https://kafka.apache.org/.
[24] "Redis," https://redis.io/.
[25] P. J. Desnoyers and P. Shenoy, "Hyperion: High volume stream archival for retrospective querying," in *USENIX ATC*, 2007, pp. 4:1–4:14.
[26] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *ACM SIGMOD*, 2014.
[27] "Apache Kudu." https://kudu.apache.org/.
[28] 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," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1379–1390, Aug. 2008.
[29] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N. Tatbul, "Flexible and scalable storage management for data-intensive stream processing," in *EDBT*. ACM, 2009, pp. 934–945.
[30] D. M., K. P., M. G., and T. M., "State access patterns in stream parallel computations," in *IJHPCA*, 2017.
[31] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri, "Macrobase: Prioritizing attention in fast data," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. ACM, 2017, pp. 541–556.