Storage and Ingestion Systems in Support of Stream Processing: A Survey

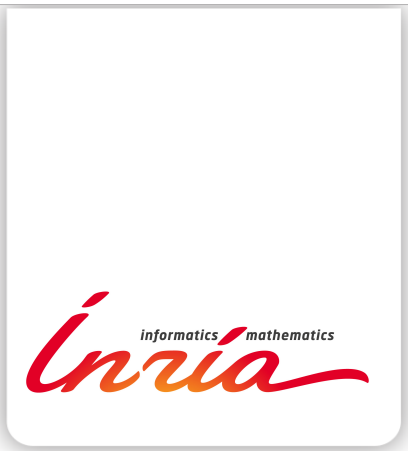
Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S Pérez-Hernández, Radu Tudoran, Stefano Bortoli, Bogdan Nicolae

Storage and Ingestion Systems in Support of Stream Processing: A Survey

Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández, Radu Tudoran, Stefano Bortoli, Bogdan Nicolae

TECHNICAL REPORT
N° 0501
November 2018
Project-Team KerData

ISSN 0249-0803 ISRN INRIA/RT--0501--FR+ENG



Storage and Ingestion Systems in Support of Stream Processing: A Survey

Ovidiu-Cristian Marcu*, Alexandru Costan†*,

Gabriel Antoniu‡*, María S. Pérez-Hernández§,
Radu Tudoran¶, Stefano Bortoli‖¶, Bogdan Nicolae**

Project-Team KerData

Technical Report n° 0501 — November 2018 — 33 pages

* Inria Rennes - Bretagne Atlantique, KerData, ovidiu-cristian.marcu@inria.fr
† IRISA / INSA Rennes, KerData, alexandru.costan@irisa.fr
‡ Inria Rennes - Bretagne Atlantique, KerData, gabriel.antoniu@inria.fr
§ Universidad Politecnica de Madrid, mperez@fi.upm.es
¶ Huawei Germany Research Center, radu.tudoran@huawei.com
‖ Huawei Germany Research Center, stefano.bortoli@huawei.com
** Argonne National Laboratory, bogdan.nicolae@acm.org

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract:
Under the pressure of massive, exponentially increasing amounts of heterogeneous data that are generated faster and faster, Big Data analytics applications have seen a shift from batch processing to stream processing, which can reduce the time needed to obtain meaningful insight dramatically. Stream processing is particularly well suited to address the challenges of fog/edge computing: much of this massive data comes from Internet of Things (IoT) devices and needs to be continuously funneled through an edge infrastructure towards centralized clouds. Thus, it is only natural to process data on their way as much as possible rather than wait for streams to accumulate on the cloud. Unfortunately, state-of-the-art stream processing systems are not well suited for this role: the data are accumulated (ingested), processed and persisted (stored) separately, often using different services hosted on different physical machines/clusters. Furthermore, there is only limited support for advanced data manipulations, which often forces application developers to introduce custom solutions and workarounds. In this survey article, we characterize the main state-of-the-art stream storage and ingestion systems. We identify the key aspects and discuss limitations and missing features in the context of stream processing for fog/edge and cloud computing. The goal is to help practitioners understand and prepare for potential bottlenecks when using such state-of-the-art systems. In particular, we discuss both functional (partitioning, metadata, search support, message routing, backpressure support) and non-functional aspects (high availability, durability, scalability, latency vs. throughput). As a conclusion of our study, we advocate for a unified stream storage and ingestion system to speed-up data management and reduce I/O redundancy (both in terms of storage space and network utilization).

Key-words: Big Data, Stream Processing, Storage, Ingestion, Data Characteristics, Stream Requirements, IoT

Storage and Ingestion Systems in Support of Stream Processing: A Survey

Résumé :

Sous la pression des quantités énormes de données hétérogènes, croissant de manière exponentielle et générées de plus en plus rapidement, les applications d'analyse Big Data ont passées du traitement par lots au traitement de flux, ce qui peut réduire considérablement le temps nécessaire pour obtenir des informations significatives. Le traitement de flux est particulièrement bien adapté pour relever les défis de fog et edge computing: une grande partie de ces données volumineuses provient d'appareils Internet des Objets et doit être acheminée en continu via une infrastructure de périphérie vers de nuages centralisés. Il est donc naturel de traiter autant que possible les données sur leur chemin plutôt que d'attendre que les flux s'accumulent sur le nuage. Malheureusement, les systèmes de traitement de flux de pointe ne conviennent pas à ce rôle: les données sont accumulées (ingérées), traitées et conservées (stockées) séparément, souvent à l'aide de différents services hébergés sur différentes machines physique/des clusters. De plus, la prise en charge des manipulations des données avancées est limitée, ce qui oblige souvent les développeurs d'applications à introduire des solutions personnalisées ainsi que des solutions de contournement. Dans cet article d'enquête, nous présentons les principaux systèmes de stockage et d'ingestion de flux à la pointe de la technologie. Nous identifions les aspects clés et discutons des limitations et des fonctionnalités manquantes dans le contexte du traitement de flux pour fog/edge et cloud computing. L'objectif est d'aider les praticiens à comprendre et à se préparer aux goulots d'étranglement potentiels liés à l'utilisation de tels systèmes à la pointe de la technologie. Nous traitons en particulier des aspects fonctionnels (partitionnement, métadonnées, assistance à la recherche, routage de message, support de contre pression) et non fonctionnels (haute disponibilité, durabilité, évolutivité, latence par rapport au débit). En conclusion de notre étude, nous préconisons un système de stockage et d'ingestion de flux unifié pour accélérer la gestion des données et réduire la redondance des I/O (à la fois en termes d'espace de stockage et d'utilisation du réseau).

Mots-clés : Big Data, Stream Processing, Storage, Ingestion, Data Characteristics, Stream Requirements, IoT

1 Introduction

The Big Data age is there: in an increasing number of areas we are flooded with massive, exponentially increasing amounts of heterogeneous data that are generated faster and faster. The age dominated by *offline* data analysis is now far behind, as data now needs to be processed fast, as it comes, in an *online* (streaming) fashion. This online dimension of data processing makes it challenging to obtain the low latency and high throughput needed to extract meaningful insight close to the data rate. To this end, Big Data analytics increasingly relies on *stream processing* as a tool that enables online analysis. In this context, online and interactive Big Data processing runtimes (e.g. Apache Flink [1], Apache Spark Streaming [2], etc.) designed for stream processing are rapidly evolving to complement traditional, batch-oriented runtimes (such as Google MapReduce [3] or Hadoop [4]) that are insufficient to meet the need for low latency and high update frequency of streams [5, 6, 7, 8, 9].

With the advent of the Internet of Things (IoT) and the immense amount of data generated by sensors and devices, the online dimension of data is further exacerbated as it passes through edge infrastructure towards public and private clouds. Unsurprisingly, this introduced a shift in stream-based Big Data processing from a traditional centralized approach on clouds towards a large-scale geographical distribution [10]. Sending all the data generated by sensors to the Cloud for processing is simply unproductive since this requires significant bandwidth from high latency networks. Instead, the load of stream processing is now pushed from the cloud data centers towards the *edge* of the network (i.e., on smart routers, intelligent gateway devices, data hubs or smart devices - known to form what are called the *Fog computing* infrastructures). The goal is to have an online/real-time front-end for processing *in the Fog*, close to the location where data streams are generated, while the Cloud will only be used for off-line back-end processing, mainly dealing with archival, fault tolerance, and further processing that is not time-critical.

This remarkable reversal of direction of the stream processing tide [11] also calls for new research on the adjacent layers of the stream analytics stack: *ingestion* and *storage*, which directly serve the processing layer. At the *ingestion layer*, stream data is acquired and eventually pre-processed (e.g. filtered), while at the *storage layer* streams are temporarily stored or durably retained for later queries. Stream processing engines consume stream elements (records or events) delivered by the ingestion and the storage layers.

In this article, we make a first step towards this load reversal, by characterizing the main state-of-the-art stream storage and ingestion systems. Our goal is **to help practitioners find the appropriate design principles and building blocks for the next-generation frameworks for stream storage and processing**. Furthermore, disruptive approaches will be needed to cope with the substantially more complicated Fog environments and to address the limitations we have found in the existing state-of-the-art. Therefore, this article aims to identify the key aspects that enable better support for Fog computing.

This study is organized as follows. We first present a set of modern applications that illustrate the need for stream processing. Stream-based applications naturally develop a need for efficient support for data ingestion and storage: stream events require not only fast ingestion, but also fast, real-time processing; for instance, stream data often needs to be indexed as it arrives, with a single pass over the data; processed stream data (in the form of filtered records or aggregated results) also requires efficient storage support, favoring low-latency, fine-grained queries (Section 3).

Next, we specifically identify the requirements that need to be addressed by the data ingestion and storage components of stream-processing frameworks: we present a set of stream characteristics and we discuss the most relevant features of data ingestion and storage systems. It is intended to serve as a reference for the users of Big Data streaming architectures, to enable

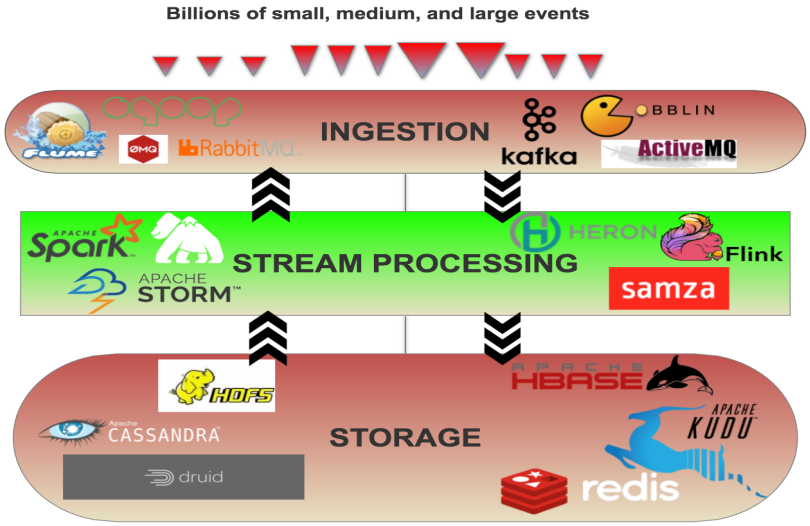


Figure 1: The usual streaming architecture: data is first ingested and then it flows through the processing layer which relies on the storage layer for storing aggregated data or for archiving streams for later usage.

them to select the appropriate tools to rely on (Section 4).

We further characterize the strengths and the limitations of state-of-the-art open-source ingestion and storage systems. Our study analyzes representative message queuing and generic frameworks for data ingestion (Section 5), and dives into key-value stores, specialized stores, time series stores, columnar stores, and structured stores as possible candidates for storage systems for stream processing (Section 6). Then we present an overview of how representative systems for stream ingestion and storage address the identified requirements (sections 5.3 and 6.6).

In particular, we discuss what storage-related features are missing to adequately support modern stream processing (Section 7). We formulate a set of takeaways (Section 8) related to future directions for ingestion and storage of stream processing, in particular with respect to the flexibility needed to cope with a broad spectrum of requirements, the architectural changes needed to enable efficient storage for data streams and how to provide better support for Fog computing. Finally, we conclude in Section 9.

2 Background: The Stream Processing Pipeline

A typical state-of-art online Big Data analytics runtime is built on top of a three layer stack (Figure 1):

- **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 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 results of stream analytics, 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/or intermediate results to the storage or ingestion layers in order to enable other streaming applications in a real-time workflow scenario or to checkpoint and later recover intermediate results; the persistent storage layer is typically based on HDFS [12], for interoperability reasons with previous developments in the area of Big Data analytics, although it was not designed for streaming patterns and cannot as such provide the required sub-second performance (e.g. online access to data repositories).

The difficulty to maintain lambda architectures (robust and fault-tolerant systems that are able to serve a wide range of workloads, e.g., support both online and offline data access patterns required by certain streaming use cases) suggests the need for *a dedicated solution for low-latency stream storage*. Such a solution should of course keep providing traditional storage functionality, but should also meet the access requirements of stream-based processing (e.g., low latency I/O access to data items or range of items).

3 Motivating Use Cases

Stream processing can solve a large set of business and scientific problems, including network monitoring, real-time fraud detection, e-commerce, etc. In short, these applications require real-time processing of stream data (i.e., an unbounded sequence of events), in order to gather valuable insights that immediately contribute with results for final users: streams of data are pushed to stream systems and queries are continuously executed over them [13]. We describe below examples of modern stream-based scenarios that exhibit challenging requirements for state-of-art ingestion and storage systems.

3.1 Monetizing streaming video content

This use case motivates the Dataflow [14] 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 with these applications:

- Events are *ingested* as fast as possible and consumed by processing engines that are updating statistics in real time;
- Events and aggregated results are *stored* for future usage (e.g. offline analysis);
- Users *interrogate* streams (SQL queries on streams, monitoring) to validate business quality agreements; this requirement introduces further challenges due to data size and stream metadata (characterized by event's attributes).

3.2 Distributed system monitoring

The log component is a first-class member of many business or scientific applications because it can deliver a concise description of the current status of a running application. There are many use cases related to real time log processing: monitoring server metrics (CPU, memory, disk, network) in order to gain insights into the system health and performance; monitoring application usage to gain insights into user activity and get real-time alerts and notifications that help maintain the service level agreements. In most scenarios data is immediately dropped after it gets filtered, while some cases require that data be retained for a certain configurable amount of time.

A similar use case is described in [15] to motivate a novel stream archiving system. Network monitoring [16] can serve for management and forensic purposes, in order to track system attacks or locate performance problems. Real-time network monitoring involves certain steps:

- Network packet headers are ingested, indexed and archived in real time;
- Monitoring systems run continuous queries on live and archived data and generate alerts if problems are identified;
- Long living stream analytics are deployed to gather more insights post-ingestion;
- Backpressure techniques are necessary to handle peak moments when data streams arrive with higher throughput;
- Streams need to be immediately ingested but only temporarily persisted until insights are extracted.

3.3 Search engines

Clustering (high-dimensional) stream data is an important problem and has many applications such as weather monitoring, stock trading, website analysis. One of the most used applications of clustering is that of search engines, which try to group similar objects (e.g. web-pages) in one cluster (e.g. search results). These applications are characterized by very large volume of data streams that require to be ingested and processed once (a single scan) in real time, after which the stream records are discarded and the summary statistics are stored for generating clusters [17].

3.4 Decision support for Smart Cities applications

Future cities will leverage smart devices and sensors installed in the public infrastructure to improve the citizens' life. In this context, several aspects are important:

- Streaming data from sensors may be initially *ingested and pre-processed at the edge*, before they are delivered to the streaming engines;
- Massive quantities of data are received over short time intervals;
- Ingestion components have to support a high frequency of stream event rates;
- Stream applications need to efficiently scale up and down based on input; this further introduces challenges for dynamic data partitioning techniques.

3.5 Social networks

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 by 40% year over year (e.g., due to social networks, 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%) [18]. 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.

3.6 Summary of stream applications features

A few general trends can be observed from the applications presented above. First, the data access model is complex and involves a diversity of data sizes, access patterns, and layouts. Second, the stream processing frameworks need to enable advanced features for the applications, such as partitioning, metatdata management, pre-preprocessing, flow control. In addition to these features, they need to address also non-functional aspects such as high availability, streaming data durability and control of latency vs. throughput. These general trends have inspired the criteria for the characterization of data streams detailed below.

4 Stream Ingestion and Storage: Requirements

Big Data streaming architectures rely on message broker solutions that decouple data sources (i.e., how stream data is ingested) from applications (i.e., how data is processed). As described in Figure 1, they are organized in three layers: a first one to efficiently acquire data, a second one to process them, and a third layer to store results or even to archive streams.

The broad question we aim to address is: *how to build streaming architectures that need to efficiently handle very diverse stream applications*? To answer this question, a critical subsequent question on which we focus in this report is: *what are the requirements that need to be addressed by the data ingestion and storage components of stream-based applications*?

In this section we focus on the characteristics of data streams and we discuss the main requirements and constraints of the ingestion and storage systems, based on which users of Big Data streaming architectures could select the appropriate tools to rely on.

4.1 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 (e.g., this study [19] shows the distribution of key-value sizes for real traces). 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). In the Fog context, ingestion and storage systems close to data sources may be tuned for *tiny* and *small* items (defined by a message length of tens of bytes (8-100B), and respectively hundreds of bytes (100-1,000B)). Storing aggregated streams to the Cloud may benefit from storage systems tuned for *medium* (a few kilobytes) and *large* (more than 1MB) items.

Data Access Patterns. Stream-based applications dictate the way data is accessed, with most of them (e.g., [20, 14]) 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 [21].

Data Model. This describes the representation and organization of single information items flowing from sources to sinks [22]. Streams are modeled as simple records (optional key, blob value for attributes, optional timestamp), messages in a queue, tuples in a key-value store or row/column sets in a table. The data model together with the data size influence the data rate (i.e., throughput, records per second), 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) should be implemented and supported by stream systems. We notice that data compression mechanisms work well with column-oriented data layouts, as we will describe later.

Data Layout. The way data is stored or accessed influences the representation of stream data in memory or on flash/disk. For example, the need to query archival stream data and the semantics of joining streams and relational tables will force the data layout to be represented in a columnar oriented approach. For some heterogeneous workloads, queries need also to update data, so a hybrid row-column layout may be necessary. It is indeed recognized that column-stores are more I/O efficient for read-only queries, leveraging techniques like push filters in order to read only the required attributes from memory or from disk [23]). Applications needing richer data types can rely on columnar formats like Apache ORC, Parquet [24] or CarbonData, or encodings such as Avro [25] (a data serialization system that provides rich data structures) which integrate within the Hadoop ecosystem and provide a statically typed interface between producers and consumers.

4.2 Functional requirements

We hereby discuss a set of critical features of ingestion and storage systems to enable continuous data stream processing (i.e. to guarantee real time, scalable, and fault-tolerant stream execution).

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. Several works study partitioning functions for stream processing that employ stateful data parallelism to improve application throughput [26]. The optimization of computation partitioning for streams between mobile and cloud may lead to increased stream processing throughput [27]. Several systems for scalable in-memory and out-of-core graph processing rely on complex streaming partitioning techniques [28].

Some stream architectures partition stream data in subspaces based on time intervals, while others leverage different techniques to implement this partitioning: consistent hashing, broker/topics (in Kafka) or segments/tables (in Druid or Kudu). We differentiate between data partitioning techniques implemented by ingestion and storage systems and application-level (user defined) partitioning [29, 30], 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 streams and it is used to easily find particular instances. How an ingestion or storage system for streaming is handling metadata is usually a fine-grained challenge and a part of design. It may be important for the system to be able to index data by certain attributes of the stream that are later used in a query. Users may want to avoid systems that limit their capabilities, for instance ones that are capable of handling only time-based metadata and not value-attribute ones.

Support for Search. The search capability is a big challenge, especially for specialized storage systems. Ingestion frameworks usually do not support such feature. A large number of stream applications [31] need APIs allowing users to scan (random or sequential) the stream datasets and to support the execution of 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 [32] for a characterization of message transport and routing.

Backpressure Support. Backpressure refers to the situation where one system cannot sustain 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. Ideally, it is advisable to avoid strategies like load shedding, where input tuples are simply dropped in order to avoid running out of memory.

4.3 Non-functional requirements

When building their stream architectures, users should first focus on verifying the stream data characteristics and second on satisfying the required features. Final decisions should not be made upon assumptions about the chosen systems. Instead, users are required to validate that their systems are production-ready, ensuring high availability and scalability. Also, stream systems should easily accommodate various persistence levels and expose the right APIs so that users could also control performance trade-offs between latency and throughput. Below, we provide a list of criteria that can help to characterize ingestion and storage systems and to identify how well a given system matches the requirements.

High Availability. This feature characterizes systems that are capable to operate continuously for a long time in spite of failures (including network partitions or hardware failures). 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 (detailed semantics and possible solutions are given in [13, 33]).

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 [34] 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 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 peak moments when high rates of bursting data arrive. Scalable stream storage should scale either vertically (increase the number of clients that are supported by a single-node broker/data node) or horizontally (distribute clients among a number of interconnected nodes). The main key to scalable stream storage is usually the stream partitioning model and its flexibility towards application-level partitioning support.

Latency versus Throughput. Some applications need low latency access to streams, while others require high throughput and can accommodate higher latencies. It is not easy to enable

System	Size	Access	Model	Compression	Layout
Apache Flume	Configurable maximum size of a single event line; Blob message handler	Record (event)	Generic	Avro event serializers	File channel backed by local filesystem or in-memory channel; HDFS sink
Gobblin	Supports Kafka topics and an HDFS writer for byte arrays; Average record size estimator of each partition to optimize work units	Record, hdfs files and splits, Kafka offsets	Generic	Avro/Orc with Hive and MR compactors	Serialized jobs and tasks states into Hadoop sequence files
Apache Kafka	Configurable message length; Optimal for small and medium messages	Record and logical offsets (each record has an offset sequential id number)	Records with key, payload attributes and optional timestamp	Relative offsets used for compressed messages; compression types supported are LZ4, GZIP, Snappy, producer, none; log compaction	Partitions as ordered, immutable sequence of records, appended to structured commit logs (on disk)
Apache ActiveMQ	Supports also blob messages	JMS Message	Queue/Topic	Message compression strategy (ZLIB)	Transactional file, in-memory based; JDBC; append-only data logs, BTree indexes
RabbitMQ	Queue length limit (number of messages or size)	JMS Message	Queue/Exchange	Serialized objects (byte array)	On-disk message store (key-value); Messages with size less than 4KB are stored in-memory in queue indexes

Table 1: Stream data characteristics for ingestion frameworks.

both low latency and high throughput, especially in some cases requiring acknowledgments. We envision that a future unified ingestion and storage system could be designed to better handle this trade-off [35].

5 Data Ingestion Systems

In Big Data stream processing architectures, the stream processing component is tightly coupled with the data ingestion component and with the storage component. The data ingestion component serves to acquire and temporarily store stream data. We differentiate between two classes of ingestion tools:

- the first class is based on *message queuing*, which basically offers interfaces for producing and consuming streams of data efficiently and in real time;
- the second class is represented by *generic frameworks* for ingesting data from various sources (e.g. databases, REST APIs, Kafka, other systems).

System	Partitioning	Metadata	Routing	Backpressure
Apache Flume	Multiple agents, sources and sinks	Record's attributes used for routing; Sources	Multi-hop agent, fan-in/fan-out, and fail-over flows; contextual routing	Flow multiplexer, channels act as buffers and propagate to the source of the flow
Gobblin	Tasks and writers partitioners	Sources	Fork operator that classifies and writes ingested data (groups) records to different places in HDFS, based on some field (classifier)	Merging and load balancing work units
Apache Kafka	Topics and partitions	Each message has a header with: format, timestamp, attributes identifier, checksum	Routing transformations with Kafka Connect (Timestamp and Regex Routers); Producer's partitioner	Pull-based consumers, buffers and replay by offsets (consumer position); Quotas on produce and fetch requests
Apache ActiveMQ	Client-side traffic partitioning, vertical and horizontal partitioning	Message ID	Wildcards and composite destinations, virtual topics, routing engine with Apache Camel	Message redelivery, producer flow control to slow the rate of messages
RabbitMQ	Sharded queues (exchanges)	Message attributes	Direct, topic, fan-out, and headers exchanges	Memory threshold will block connections that publish messages

Table 2: Functional requirements for ingestion frameworks.

5.1 Message queuing systems

Apache Kafka [36, 37] is a distributed stream platform that provides durability and publish/subscribe functionality for data streams (making stream data available to multiple consumers). It is the *de facto* open-source solution used in end-to-end pipelines with streaming engines like Apache Spark or Apache Flink, which in turn handle data movement and computation. A Kafka cluster is a set of one or more broker nodes that store streams of records in categories called topics. Each topic can be split into multiple *partitions* (allowing to logically split stream data and parallelize a topic) that are managed by a Kafka broker (i.e., a service collocated to the node storing the partition that further enables consumers and producers to access the topics data). Apache Kafka gets used by producer services that are required to put data into a topic (Producer API), consumer services that can subscribe and process records from a topic (Consumer API) or applications that can consume input streams from one or more topics and can produce output streams to one or more output topics (Streams and Connector APIs).

Apache ActiveMQ [38] is a message broker that implements the Java Message Service (JMS) specification and provides high availability, scalability, reliability and security for enterprise messaging. ActiveMQ provides many connectivity options, including support for protocols such as HTTP/S, SSL, STOMP, TCP, UDP, XMPP, and IP multicast. There are two styles of messaging JMS domains: PTP (point-to-point) uses queues destinations through which messages are sent and received either synchronously or asynchronously, each message being delivered exactly once to a consumer; pub/sub (publish/subscribe) uses topics destinations where producers send messages and subscribers register either synchronously or asynchronously in order to receive messages - a topic can be configured as a durable subscription in order to retain messages in the event a subscriber disconnects. PTP does not guarantee message durability. ActiveMQ supports two types of message delivery: a persistent message is logged to stable storage while a non-persistent message is not guaranteed to be delivered. ActiveMQ uses a push-based model to

deliver messages to consumers.

RabbitMQ [39] is a message broker that implements the Advanced Message Queuing Protocol (AMQP [40]) for high-performance enterprise messaging. The AMQP model is based on *exchanges*; an exchange is a message routing agent that is responsible for routing messages to different queues. Instead of publishing messages directly to a queue, a producer sends messages to an exchange. Binding links are used to bind an exchange to possibly multiple queues and a routing key (message attribute) is used to decide how to route the message to queues. It is possible to define different exchange types: a *direct exchange* delivers messages to a queue if the binding key matches exactly the routing key of a message, a *topic exchange* is routing messages based on routing keys matching a routing pattern specified by the queue binding, and a *fan-out exchange* simply copies and routes a message to all bounded queues, ignoring the routing keys or binding patterns. Consumers can be configured to receive messages (push api) or pull messages from queues on demand (pull api).

Other queuing systems offer similar functionality. *ZeroMQ* [41] is an embeddable concurrency framework for asynchronous messaging and does not require a dedicated message broker. ZeroMQ offers support for direct and fan-out endpoint connections and it implements different messaging patterns like request/reply, pub/sub, push/pull. *HornetQ* [42] is a message-oriented middleware that implements the JMS specification. HornetQ can be run as a stand-alone server, an embeddable container and as a cluster, offering server replication and automatic client failover. HornetQ offers support for sending and receiving very large multi-gigabyte messages and provides support for backpressure by implementing producer and consumer rate limited flow control.

5.2 Generic frameworks

Apache Flume [43] is used for efficiently collecting, aggregating and moving large amounts of event data (e.g., network traffic data, social media, email messages) from various sources to centralized data stores. Each Flume agent is a JVM process that hosts source, channel, and sink components through which events flow from an external source to the next destination. A durable channel is backed by a file (channels can be backed by an in-memory queue but events will be lost if the agent process dies) and is responsible for temporarily storing events that are eventually consumed by a sink and further put into an external repository like HDFS.

Gobblin [44, 45] is a batch-based generic data ingestion framework for Hadoop HDFS. It constructs workflows (each job can include a source, an extractor, a converter, a quality checker, a writer, and a publisher) in order to ingest data from various sources such as relational stores, streaming systems, rest endpoints, file systems, Kafka topics. Gobblin supports a record-level abstraction and plans to add support for file-level. Its backpressure mechanism is based on two techniques: merging work units in order to reduce the number of small files published to HDFS and load balancing of work units to containers.

Other generic frameworks focus on some specific functionalities. *Elasticsearch* [46] is a distributed real-time search engine designed to ingest and store JSON documents, based on Logstash for ingesting and transforming data on the fly; it uses Apache Lucene to handle indexing and stores files and metadata on disk in custom data directories. It integrates with Hadoop, with which users can build dynamic, embedded search applications in order to serve HDFS-based data or can perform low-latency analytics using full-text, geospatial queries and aggregations. Elasticsearch has the ability to index data on timestamp fields and provides primitives for storing and querying time-series data. *Apache Sqoop* [47] is a tool designed for efficiently transferring bulk data between HDFS and (semi-) structured data stores (e.g. HBase, Cassandra, relational databases).

System	Availability	Scalability	Persistence	Latency	Throughput
Apache Flume	Supports channel replication	Targets horizontal scalability by adding more agent nodes (multi-agents)	Durable file channel (on-disk), temporary in-memory channel	Configurable sink batch size, trade-off with throughput	High if used an in-memory queue
Gobblin	Given by Hadoop, retries failed tasks	Hadoop MapReduce on a Hadoop cluster	Checkpoints state on disk (uses Hadoop sequence files)	Low, data can be registered to Hive right after it's published at destination	High, Limited by disk (HDFS)
Apache Kafka	Replicated partitions, configurable acknowledgements to allow trade-offs with durability	Multi-Broker horizontal scalability	Temporal (data retention policies), on-disk	Low, depends on acknowledgements	High, batching size on producer, reduced for small messages due to additional timestamp; high (MBs) or low (number of records) for large messages
Apache ActiveMQ	Shared-nothing and shared-storage master/slave	Network of brokers topologies	File-based, memory or JDBC store	Depends if message is persistent or dispatched asynchronously	Higher with larger pre-fetch sizes and setup of acknowledgements in batches
RabbitMQ	Exchanges, bindings, queues are mirrored to replicate their contents	Clustering and federation	Persistent/transient messages	Consumer prefetch; Channel limits	Higher if acknowledged multiple messages

Table 3: Non-functional requirements for ingestion frameworks.

5.3 Overview of ingestion systems for streaming

In Table 1 we analyze ingestion frameworks according to five important data characteristics. We observe that ingestion tools are designed to handle records of different sizes (enabling **fine-grained access**), however the available documentation does not make it clear what optimizations are used to efficiently store streams. Each system is able to implement different compression mechanisms and there is no preferred storage system used for temporarily storing data. Compared to other messaging systems, **Kafka seems to relax consistency in favor of better performance** (e.g., throughput). When evaluating their stream use case, users should first assess the **data size** and **access characteristics** in order to understand whether the selected ingestion framework is able to handle optimally their data.

In Table 2 we describe how ingestion systems handle the various specific challenges of data streaming that we discussed in Section 4.2. In general, **search is a non-goal**: *Gobblin* allows the specification of patterns on HDFS datasets and Hive partitioned tables; *RabbitMQ* maintains a queue index responsible to maintain knowledge if a given message is in a queue, provided that it has been delivered or acknowledged. We observe that ingestion frameworks offer **good support for routing and backpressure**.

In Table 3, we summarize how ingestion systems address non-functional requirements related to the criteria described in Section 4.3. When selecting an ingestion framework, users should validate the latency-throughput trade-offs considering the required consistency semantics (e.g., ensuring exactly once processing may require acknowledgement of data and durable storage, limiting throughput and thus increasing latency).

6 Storage Systems for Streaming

In a Big Data stream processing architecture, a storage layer is necessary in two cases: to archive stream data for later usage (e.g. continuous queries [48]), or to manage persistent local (intermediate) state (e.g., when processing large windows that span months, maybe years of data). Based on the application characteristics (e.g. access patterns, data model), storage systems can fall into one of the following categories:

- key-value stores (offering fine-grained access, e.g., put/get by key);
- specialized stores for streaming systems (graph-based stream applications);
- time series databases (specialized for IoT applications);
- structured stores (supporting range queries);
- columnar stores on top of Hadoop HDFS.

6.1 Key-value stores

Redis [49] is an in-memory data structure store that is used as a database, cache and message broker. Redis supports many data structures such as strings, lists, hashes, sets, sorted sets, bitmaps and geospatial indexes. Redis implements the pub/sub messaging paradigm and groups messages into channels with subscribers expressing interest into one or more channels. Redis implements persistence by taking snapshots of data on disk but it does not offer strong consistency [50]. Redis keys and strings are binary safe (can contain any kind of data, e.g., an image) and their value can be at max 512 MiB. In Redis, many data types are optimized to use less space (data encoding, CPU/memory tradeoff) if their size is smaller than a given number of elements.

RAMCloud [51] is an in memory key-value store that aims for low latency reads and writes, by leveraging high performance Infiniband-like networks. Durability and availability are guaranteed by replicating data to remote disks relying on batteries. Among its features we have fast crash recovery, efficient memory usage and strong consistency. Recently it was enhanced with multiple secondary indexes [52], achieving high availability by distributing indexes independently from their objects (independent partitioning).

MICA [53, 54] is a scalable in-memory key-value store providing consistently high throughput over a variety of mixed reads and writes workloads, while using a single general-purpose multi-core system. MICA enables parallel access to partitioned data and adopts a lightweight-networking stack that bypasses the kernel (directly interfacing with NICs): systems using standard socket I/O, optimized for bulk communication, incur high network stack overhead at both kernel and user-level. MICA is designed to achieve high single-node throughput, consistent performance across workloads, striving for low end-to-end latency on commodity hardware and handling small, variable-length key-value items.

HyperDex [55] is a distributed key-value store that provides search primitives in order to enable (range) queries on secondary attributes, additionally to single object retrieval by its primary key. HyperDex organizes its data through a technique called hyperspace (multi-dimensional Euclidean space) hashing, that deterministically maps objects to servers (taking into account secondary attributes) in order to allow for efficient inserts and searches. HyperDex provides strong consistency (and fault-tolerance for concurrent updates) by leveraging a technique in which updates to objects are ordered into a value-dependent chain with members deterministically chosen from an object's hyperspace coordinate. HyperDex offers both key and search consistency to guarantee a search will return objects committed at search time.

Other streaming engines have various needs for caching internal state (in order to speed-up certain operations) and they rely either on specialized caching systems such as *Memcached* [56, 57], on embedded key-value stores such as *RocksDB* [58] or *LMDB* [59].

6.2 Specialized stores

DXRAM [60] is a key-value store that keeps data always in RAM and is designed for billions of small data objects (16-128 bytes) needed by graph applications. DXRAM replicates its data using an asynchronous logging on remote disks and designed an SSD-aware logging technique in order to optimize read and write throughput, being customized for small key-value tuples.

Hyperion [15] is a system for archiving, indexing, and on-line 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. Hyperion is designed to support queries (no support for fine-grained access, limited support for range and rank queries), it exploits sequential, immutable writes (in order to reduce disk seek overhead and to improve system throughput) and its write optimized stream file system supports records not files.

Druid [61, 62] is an open-source, 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. A Druid cluster consists of specialized nodes: real-time nodes (that maintain an in-memory index buffer for all incoming events, regularly persisted to disk) give functionality to ingest and query event streams; historical nodes give the functionality to load and serve immutable blocks of data which are created by real-time nodes; broker nodes act as query routers to historical and real-time nodes (metadata is published in ZooKeeper); coordinator nodes which are in charge of data management and distribution of historical nodes (loading new data, dropping old data, data replication, load balancing). Druid's data model is based on data items with timestamps (e.g. network event logs). As such, Druid requires a timestamp column in order to partition data and supports low latency queries on particular ranges of time.

Other specialized stores focus on specific types of queries and access patterns. [63] highlights the need to offer *support for data persistence* to continuous-query [64] streaming processing systems. The authors propose a stream storage design built on top of persistent stream objects (PSO, a persistent image of the state of a queue over some time), a key concept of the proposed architecture, and evaluate it using the PVFSs [65].

In [66], a framework for data-intensive stream processing is built based on the information about the access patterns of streaming applications that are leveraged for tuning and customizing the storage component. The authors target applications that need to manage a large state during stream processing (e.g. internet traffic analysis, log monitoring/mining, scientific data processing) and try to identify a set of tunable parameters in order to define a general-purpose storage management interface. However, they do not consider distributed settings and their evaluation is limited to the Linear Road Benchmark [67].

VOLAP [68] is a scalable real-time system that supports real-time querying of high velocity data, and allows for ingestion of data, but no deletion. VOLAP is interesting as it acknowledges the need to partition data based on chosen dimension (attribute) of data, without limiting the system to a special dimension (like timestamp).

PTS [69] provides the persistent temporal stream abstraction to enable ingestion, processing and storage of stream data. PTS is designed for video/audio heavyweight streams and implements an abstraction for accessing live and historical stream data. In PTS each item has a timestamp,

a temporal stream is represented by a channel holding a time-indexed sequence of discrete data items (e.g. video frames). Applications interact with channels by using put/get or get_interval interfaces to access data.

Liquid [70] is a two-layer data integration stack for backend systems. The processing layer supports stateful stream aggregations and is based on Apache Samza [71]. The messaging layer offers data access based on metadata such as timestamps being based on Apache Kafka [36]. Liquid is an alternative to current MapReduce/HDFS stacks (used to clean and normalize data), motivated by known limitations: higher latencies access as a consequence of writing to HDFS the intermediate results of MapReduce jobs, lack of fine-grained access to data and incremental state management (to avoid reprocessing an entire job when new data is available).

Freeze-Frame File System (FFFS) [72] builds on top of HDFS in order to be able to accept streams of updates from real-time data sources while it offers support for parallel read-only access (temporal reads). FFFS gives access to consistent snapshots (granularity of tens of milliseconds) necessary for temporal analysis while computation runs in parallel. It leverages RDMA hardware if available and shows highly optimized read and write throughput for a single client. FFFS extends HDFS by re-implementing the snapshot mechanism in order to escape from a number of limitations: the append-only characteristic limits users from having random write access; users are only able to read a file after the written file is closed, frequent file close and open operations are a costly overhead and should instead be avoided. One limitation of FFFS is its missing support for replicating data.

6.3 Time series stores

TimescaleDB [73] is an open source database designed for scaling SQL for time-series data. TimescaleDB is developed on top of Postgres as an extension, providing full SQL interface. Its main challenge is to support high ingest rates for time-series data and it makes the following assumptions: time-series data are immutable and writes mainly occur as new inserts to recent time intervals, and workloads are naturally partitioned across time and space.

Other time-series data stores have been proposed with similar functionality and different stregths. *InfluxDB* [74] is an open source platform designed for time-sensitive data (e.g. metrics, events). InfluxDB stores its data on disk (time structured merge tree) and provides a SQL-like query language based on JSON over HTTP. In a benchmark which measures data ingest performance (values/second), on-disk storage requirements (MBs, compression) and mean query response time (milliseconds), InfluxDB outperforms Elasticsearch by up to 10x. Its open-source version does not support clustering, for high availability or horizontal scalability a commercial solution is available.

RiakTS [75] is an open-source enterprise nosql time series database that is optimized for IoT and time series data.

OpenTSDB [76] is an open-source scalable time series database that runs on Hadoop and HBase.

6.4 Structured and semi-structured stores

Apache Kudu [77] 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 (these are not suitable for updating individual records or for efficient random reads or writes) or an alternative to semi-structured stores like HBase or Cassandra (which are a fit for mutable data sets, allowing for low latency record-level reads and writes, but are not efficient for sequential read throughput needed by applications like machine learning or SQL). Kudu's cluster is a set of tables; each table has a

System	Size	Access	Model	Layout
Redis	Up to 512 MB. Optimized data types (e.g. lists/sets up to a certain size are encoded for memory efficiency)	Fine-grained and multi-group queries	Key-Value, data structures	In-memory backed by disk (custom checkpoints)
RAMCloud	One object has a variable 64KB key, 64-bit version number, and up to 1MB variable-length byte array value	Fine-grained and multi-group queries	Key-Value, tables	Log-structured, in RAM and on disk
Mica	Tiny, Small	Fine-grained	Key-Value	In-memory cache and store modes
HyperDex	Small, Medium. Depends on number of attributes	Fine-grained, Search by attributes, Range queries	Key-Value, structured with zero to many attributes	File-based logs on-disk
DXRAM	Tiny	Fine-grained	Key-Value, tuples as block of chunks	RAM layout (large virtual memory block), On disk (complex log architecture)
Hyperion	Small	Queries	Immutable stream records (timestamp attributes are associated with ranges in a stream)	On-disk log-structured, small fixed length blocks of 1MB
Druid	Timestamp, dimension, and metric columns	Queries (search, time-series, time boundary)	Data tables (time-stamped events)	Segments stored as columns
Apache Kudu	Typed columns (binary up to 64KB)	Fine-grained, scan (predicates, projection), queries (time-series, granular)	Tables	Columnar (memory and disk row sets)
Apache HBase	A cell is a (row, column, version) tuple of uninterrpreted bytes	Fine-grained, scan	Tables	Columnar, write ahead logs on disk in Hadoop
Apache Cassandra	CQL data types, Manages also blobs up to 2GB	Fine-grained (table, key), CQL	Tables (distributed multi dimensional map indexed by a key)	Commit log on disk, in-memory off-heap row cache
TimescaleDB	Timeseries data	Queries	Hypertable	N/A

Table 4: Stream data characteristics for storage systems.

well-defined schema and consists of a finite number of columns. A column is defined by a type and defines a primary key, but no secondary indexes. A table can be split in segments that are called tablets. Clients communicate with a tablet server. Metadata is kept in a central location called the catalog table that is accessible only via metadata operations exposed in the client API. Metadata defines table schemes, locations, tablets, and which tablet servers have replicas on each tablet. Kudu offers fast columnar scans (comparable to Parquet, ORC) and low-latency random updates. Kudu does not offer any multi-row transactional APIs and each write operation must go through the tablet's server.

Apache HBase [78] is a real time store that supports key indexed record lookup and mutation (random real time read/write access to Big Data). HBase is a distributed, versioned, non-relational database modeled after Google's BigTable [79]. One study at Facebook [80] tries to answer if HDFS is an effective storage backend for HBase, using a messaging system that enables Facebook users to send chat and email-like messages to one another (Facebook Messages) and emphasize on the importance of optimizing writes in layered systems: logging, compaction, replication and caching combined can change dramatically the read/write ratio(s).

Apache Cassandra [81, 82] is a distributed storage system for large amounts of structured data spread across many commodity servers. It is designed to provide high availability (with no single point of failure) and durability (by means of replication). Cassandra does not support a full relational data model. A table is a distributed multi dimensional map indexed by a key. The APIs consist of three methods: insert (table, key, rowMutation), get (table, key, columnName) and delete (table, key, columnName), and the Cassandra Query Language (CQL). Cassandra is able to scale incrementally, by dynamically partitioning the data over the set of nodes (consistent hashing is used to partition data).

Columnar formats

Apache CarbonData [84] is a fully indexed columnar and Hadoop data store for processing fast analytical queries on Big Data streams. Compared to traditional columnar formats, in CarbonData the columns in each row-group are sorted independently of other columns. CarbonData is designed for various data access patterns like sequential, random and online analytical processing (low latency fast queries on fast data) and has built-in integration with interfaces for Spark SQL, with support for bulk data ingestion.

Apache Parquet [85] is a columnar storage format for Hadoop, providing complex nested data structures. It implements techniques described in Dremel [24], being built to support efficient compression and encoding schemes. Parquet's default compression is Snappy. It has very good support in Apache Spark SQL.

Apache Orc [86] is a columnar format for Hadoop that includes support for ACID transactions and snapshot isolation. Orc's default compression is ZLIB. Orc has very good support in Apache Hive.

6.5 In-memory transactional databases

In-memory database systems are an important solution to support transaction management on stream applications. Although transactions are not a focus of this study, we would like to refer the user to this study [87] which provides a comprehensive review of memory management design principles and practical techniques for high-performance in-memory storage systems.

6.6 Summary: adequacy for stream processing

In Table 4 we comment on how storage systems handle stream data characteristics. In general, **compression** is a **non-goal** for key-value or specialized stores, however certain systems offer

System	Partitioning	Metadata	Search	Routing
Redis	Hash Slots (Cluster mode)	Keys	Multi-get queries	Key hashtag
RAMCloud	Tablets	Table ids, Keys	Multi-read, enumerate-Table	Split and migrate tablets
Mica	Keyhash-based. One partition consists of a single circular log and lossy concurrent hash index (cache mode). The store mode uses segregated fits [83].	Keys	Index for high-speed read and write	N/A
HyperDex	Data subspaces (dimensions, hyperspace hashing)	Record attributes and keys	Deterministic (hyperspace mapping)	Assigns regions to servers
DXRAM	Chunks and peers	Super-Peer Overlay Structure, OID tables	Node Lookup, OID search, range-based queries	N/A
Hyperion	Streams partitioned into time intervals	Record and headers, block maps, one file system root	Multi-level signature indices	N/A
Druid	Segments (shards data by time)	Segments and configuration	JSON over HTTP, SQL	JSON IO Config, Stream push/pull ingestion, batch ingestion
Apache Kudu	Tablets (range, hash, multilevel partitioning)	Catalog table (tables, tablets)	SQL	N/A
Apache HBase	Tables are split into chunks of rows called regions	Tables, column family names	Get single row, scan many rows	Region split policy
Apache Cassandra	Consistent hashing	Tables, keys, columns	Get, CQL	Partitioner, Initial-Token
TimescaleDB	Time/space partitioning, hypertable, chunks	Tables	SQL	N/A

Table 5: Functional requirements for storage systems.

System	Availability	Scalability	Persistence	Latency	Throughput
Redis	Sentinel HA	Multiple Redis Server instances (vertical), Cluster mode (horizontal)	Durable, in-memory backed on disk	Lower if used as cache (but does not benefit from multiple cores)	Higher with pipelining (multiple commands at once)
RAMCloud	HA with master/coordinator crash recovery across many nodes	Horizontal (collection of commodity servers managed by a coordinator)	Durable, data in memory with copies on disk	Low latency by kernel bypass (NIC device registers are memory mapped) and polling (busy waiting)	Not optimized for high throughput (the dispatch thread is the bottleneck)
Mica	Non-goal	Scales well with more CPU cores and more network bandwidth	Non durable, data stored in-memory	Low end-to-end latency	High single-node throughput
HyperDex	Replicates (protocol value-dependent chaining)	Horizontal (adding servers)	Durable, on-disk	Low reads/writes, increases linearly by number of subspaces	High, decreases proportionally to the number of subspaces
DXRAM	Reliable super-peer overlay network	Range-based tree structure allows fast node lookups	Durable, in-memory backed on disk	Low, random data access, read operations	SSD-aware logging to maximize read/write throughput
Hyperion	Non-goal	Multi monitors	Durable, on-disk	Dominated by per-record overhead	Exploit sequential, immutable writes
Druid	Historical nodes respond to query requests even for Zookeeper outages	Separated components (historical, coordinator, broker, indexing service, and real-time nodes)	In memory and persisted indexes on real time nodes	Data ingestion latency depends on data complexity (dimensions, metrics, aggregations)	Number of events ingested and made query-able by a real time node; throttle ingestion to prevent out of memory
Apache Kudu	HA with Raft consensus	Tablet servers and masters	Durable, on-disk	Efficient random access (delta stores are a bottleneck for certain workloads)	Lazy materialization, delta compaction, predicate push-down
Apache HBase	Region replication (HA reads)	Region Servers handle a set of regions	In-memory blocks, Durable, on-disk	Bloom filters improve read latencies	Caching settings
Apache Cassandra	Replication strategies (simple strategy for single data center and network topology for multiple data centers)	Virtual nodes (many tokens/regions per node)	Durable, on-disk	Tradeoffs consistency latency	Compression, compaction, off-heap memory management helps
TimescaleDB	Postgres HA	Horizontal	Durable, on-disk	Chunks of same region partition keyspace are collocated; Reduced tail latency for single object space	High data write rates (batched commits, in-memory indexes)

Table 6: Non-functional requirements for storage systems.

support or internally implement optimizations for compressing data. *Redis* exposes certain data encoding configurations for optimizing memory of its types. *Druid* implements strongly typed columns and uses dictionary encoding for string compression, LZF for numeric columns. *Kudu* uses pattern-based compression on its typed columns (plain, bit-shuffle, run length, dictionary, prefix encodings), and it also applies per-column compression using LZ4, Snappy, zlib. *HBase* enables different compression algorithms on a column family: data block encoding to limit duplication of information in keys and block compressors such as Snappy, LZO, LZ4, GZIP. We observe that some systems handle **data** as being **immutable** in order to optimize the write throughput (simplified data model) and offer sometimes-specialized APIs for querying data.

Some use cases (e.g., 3.2, 3.4) may require a separate ingestion component with simple processing capabilities that will pre-aggregate stream records before passing summary statistics to processing engines. Other use cases (e.g., 3.1, 3.3 3.5) may benefit from a potential unification of ingestion and storage components to help indexing data that is later queried. Use cases as described in section 3.2 may need just a simple ingestion component with filter capabilities and possibility to raise real-time alerts. Although these systems are designed to handle *records of different sizes*, it is not clear what optimizations are used to efficiently process and store them. 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 optimally handle 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 5 we review the same streaming challenges, now targeting several storage systems and without considering backpressure, which is usually a non-goal. Storage systems implement different strategies for data partitioning (time-based, key-based, or custom solutions); **what is still 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 application rules, sharing the partitioning strategy at both ingestion/storage and processing layers).

In Table 6 we provide a brief characterization of the storage systems through a set of keywords aiming to help users to better assess each tool before deciding which one(s) to consider in a future streaming solution. For a solution that offers both ingestion and storage support, the challenge is to offer **search primitives on real-time ingested data** and to provide support for **backpressure**, inspired by techniques already developed by ingestion frameworks. We observe that in order to ensure high availability, **many systems integrate with HDFS**, as the de facto system for replicating data. 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**.

7 Missing Storage Features for Stream Processing

Most of the previous systems can already manage workloads with **fine-grained and sequential access**. Their design allows for **queries** on recent (near real time) archived data. However, we identify a set of storage 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 (use cases 3.1, 3.2, 3.3, 3.4, 3.5). Streaming queries or SQL on streams have recently emerged as a convenient abstraction to process flows of incoming data, inspired by the deceptively simple decades-old SQL approach. However, they extend beyond time-based

or tuple-based stream models [88]. 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. Computing **aggregate functions** on streams is possible using various **windowing constructs** (e.g., tumbling, sliding, session windows) which require fine-grained access to data; this gets more complicated when two streams join or a stream and a table join. Even more, queries that apply DML syntax (insert, update, delete) on streams require support for mutable data streams.

Access pattern detection (use cases 3.1, 3.2, 3.3, 3.4, 3.5). Ingestion/storage for streaming would both benefit from **detecting and then dynamically adapting** to the observed stream access patterns [66, 89]: ranging from fine-grained per record/tuple access to group queries (multi get/put) or scan-based.

Streaming machine learning (use cases 3.1, 3.4, 3.5). Applying machine learning on historical and real-time data has recently become hot research topic. It is not yet clear how one can develop **mirrored stream-primitives** at the stream storage layer in order to support complex machine learning workloads (e.g., clustering evolving data streams) [90, 17, 91].

Streaming graphs (use cases 3.3, 3.5). Processing massive graphs in the data stream model [92] requires support from the stream storage system in order to **materialize dynamic complex data structures** and implement **custom partitioning** techniques.

Windowing (use cases 3.1, 3.2, 3.3, 3.4, 3.5). A basic primitive in streaming is windowing [34] 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**. Other stream operations require the management of a **fault-tolerant local state** which is usually done by checkpointing to local or distributed file systems: we need support for optimizing various read and write access patterns of the local state for multi stream operators.

Pushing user-defined aggregate functions to storage (use cases 3.2, 3.4). 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 (including multi-core fat nodes with large memories). 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.

Support for geographically distributed replication (use case 3.4). Multi-fog or hybrid fog-cloud replication should be developed similarly to multi-datacenter replication [93] in order to provide high service availability (including geo-wide fault-tolerance and read locality) in spite of fog or datacenter failures.

8 Future Directions for Ingestion and Storage for Stream Processing: Discussion

As stream processing needs to cope with increasingly larger data volumes and smaller latencies, the underlying storage layers need to be able to keep up with this pace. Distributed storage systems serving stream engines have evolved from a few very basic early systems to a myriad of modern and more general frameworks like Kafka and Kudu. We believe there is a large room for improvements at the storage layer in order to better serve the needs of today's stream-based applications.

8.1 Flexibility: coping with a large variety of requirements

The ingestion component should be designed to **efficiently handle records of different sizes (enabling fine-grained access)**, implement different compression mechanisms, support multi-tenants, and handle heterogeneous streams.

Stream partitioning techniques are complex and diverse, influenced by a variety of stream access patterns [66, 89]. Ingestion and storage systems for streaming need to provide **specialized storage primitives to support efficient implementation of stream partitioning techniques**: they are needed to support various user-defined policies for static and dynamic application-level partitioning.

As one can expect, there is **no single solution that fits every use case**; architects need to invest time to evaluate how current systems respond to data characteristics, to functional and non-functional requirements. Tables 1,2,3,4,5,6 summarize how several representative systems for ingestion and storage handle various requirements. In Section 6 we have briefly described other related systems without extending the discussion as we were limited by poor documentation or because they are not dedicated solutions for stream ingestion and/or storage.

Processing live data alone is often not enough: many applications **need to combine live data with previously archived data to increase the quality of the extracted insights**. As seen from this study, **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*.

8.2 Dedicated streaming storage architectures

As the need for more complex online data manipulations arises, so does **the need to enable better coupling between the ingestion, storage and processing layers**. Emerging scenarios (Section 3) 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 sent 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, when collecting data streams and writing archival data simultaneously). 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. We argue that the aforementioned challenges are significant enough to offset the benefits of specializing each layer independently of the other layers [94]. We consider that a plausible path to follow to alleviate from this is designing a **unified ingestion and storage architecture** for streaming data.

Only a few systems can manage both ingestion and storage, but they are limited to specific use cases (e.g. see Druid, Kudu, Hyperion) and do not try to tackle I/O redundancy and interference. We argue that a unified ingestion and storage architecture should ideally consider all the requirements and optimize for stream data characteristics. At the same time we acknowledge that it can be challenging to deploy **the same storage-ingestion solution for various use cases, but tuned for their respective needs**.

Stream processing engines should ideally focus on the operator workflow: how to transform data (define and trigger the computation) through a workflow composed of stateful operators, and how to combine the computation with offline analytics. Processing engines could avoid complicated mechanisms to store (large) processing state at this level if specialized storage

for streaming offers appropriate support. Ingestion-Storage solutions for streaming could then **focus on high-level data management and movement**, being exposed through a common engine able to leverage synergies to avoid I/O redundancy and I/O interference arising when using independent solutions for the two aspects.

A critical requirement for future storage architectures is the efficient support for **search primitives on real-time ingested data** and for **backpressure**, inspired by techniques already developed by dedicated ingestion frameworks.

Integration in the Big Data Ecosystem: the Case of HDFS. Streaming architectures evolve in large and complex systems in order to accommodate many use cases. Ingestion tools are connected either directly or by streaming engines to the storage layer. The Hadoop Distributed File System (HDFS) [12] provides scalable and reliable data storage (sharing many concepts with GoogleFS [95]), and is recognized as the de-facto standard for Big Data analytics storage. Although HDFS was not designed with streams in mind, many streaming engines (e.g., Apache Spark, Apache Flink, etc.) depend on it (e.g., HDFS sinks). Among HDFS limitations, we recall the metadata server (called NameNode) as a single point of failure and a source of limited scalability, although solutions exist [96] to overcome the metadata bottleneck. Also, there is no support for random writes as a consequence of the way the data written is available for readers (only after the file is closed, data can be appended to files). In fact, in [97] authors point out that HDFS does not perform well for managing a large number of small files, and discuss certain optimizations for improving storage and access efficiency of small files on HDFS. This is why streaming engines develop custom solutions to overcome limitations of HDFS.

Given the fact that most of the ingestion and stream storage systems along with streaming engines integrate with Hadoop HDFS, **a unified solution considering HDFS as storage for historical streams** has a strong appeal. Its implementation could consider a message queuing system like ZeroMQ for stream ingestion and the development of a monitor component similar to Hyperion's that will share the resources with a DataNode for indexing stream metadata. In addition, a set of columnar formats are developed and integrated with Hadoop (HDFS), however **HDFS was not designed to store small data or billions of files**. An in-depth evaluation and comparison of these formats for certain benchmarks would be useful. This required study should help responding to important questions:

- How can HDFS be extended to support billions of small items?
- How can HDFS be tuned to enable high performance data access on large collections of small items (e.g., ms I/O access for scan, range queries and random access)?
- What are the best partitioning techniques, data placing and search strategies for stream data storage on HDFS?

8.3 Providing better support for edge/fog computing

Edge devices are limited storage/computational resources where time-sensitive IoT streams are initially ingested and processed, and later aggregated streams are passed to Cloud for exploring other insights. Because edge resources are limited, **a unified storage and ingestion architecture for streams would reduce deployment costs and reduce or eliminate I/O redundancy on edge/fog systems**.

Architects should consider to **expose latency-throughput trade-offs for different consistency levels** (exactly once, at least once, at most once), and apply accordingly the necessary performance optimizations e.g, to increase throughput, decrease latency; ensuring exactly once

processing requires acknowledgement of data and durable storage, possibly limiting throughput). As such, fog stream storage solutions could be tuned for low-latency processing while cloud stream storage solutions could be tuned for high-throughput processing.

Cloud and fog computing [98] collaborate to effectively deploy Smart applications [99] and introduce a set of additional requirements that need to be addressed by a unified storage and ingestion architecture:

- Multi-protocol support for edge-level ingestion and batch aggregation techniques to communicate with Cloud and distributed edge components to overcome network links capacities;
- Stream data heterogeneity for both data content and the number of streams;
- Pushing stream-based computation to fog stream storage on demand.

9 Conclusion

This paper provides an overview of state-of-art systems for data ingestion and storage for stream processing. Stream-based processing appears as particularly adequate as more and more data is generated by Internet-of-Things (IoT) devices and require efficient extraction of relevant knowledge. This efficiency becomes even more critical as processing architectures tend to combine edge/fog infrastructures with "big" and powerful datacenters.

Our study provides an extensive analysis of how they address functional applicational requirements (partitioning, metadata, search support, message routing, backpressure support) as well as non-functional aspects (high availability, durability, scalability, latency vs. throughput) of stream-based applications. The goal is to help practitioners understand and prepare for potential bottlenecks when using such state-of-art systems.

Our analysis identifies multiple limitations and missing features of state-of-the-art systems in the context of stream processing, which are particularly relevant in the context of the emergence of fog computing. Therefore, we plead in favor of a *unified stream storage and ingestion* system to speed-up data management and reduce I/O redundancy (both in terms of storage space and network utilization). Given the fact that most of the ingestion and stream storage systems along with streaming engines integrate with Hadoop HDFS, such a unified solution considering *HDFS as storage for historical streams* appears as an appropriate direction. In future work, we intend to deepen this analysis and develop detailed design principles for such an architecture as well as a proof of concept.

Acknowledgments

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

References

[1] “Apache Flink.” http://flink.apache.org.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[3] 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.

[4] "Hadoop," https://hadoop.apache.org.

[5] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *Proceedings of the 10th International Conference on Database Theory*, ser. ICDT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 37–52.

[6] "The world beyond batch: Streaming 101," https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

[7] "The world beyond batch: Streaming 102," https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102.

[8] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00778-004-0147-z

[9] 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 *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1087–1098. [Online]. Available: http://doi.acm.org/10.1145/2882903.2904441

[10] G. Fox, S. Jha, and L. Ramakrishnan, *STREAM2016: Streaming Requirements, Experience, Applications and Middleware Workshop*, Oct 2016. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1344785

[11] S. Banerjee and D. O. Wu, "Final report from the nsf workshop on future directions in wireless networking," USA, Tech. Rep., 2013.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[13] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2005.72

[14] 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, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076

[15] P. J. Desnoyers and P. Shenoy, "Hyperion: High volume stream archival for retrospective querying," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ser. ATC'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 4:1–4:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364385.1364389

[16] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 647–651. [Online]. Available: http://doi.acm.org/10.1145/872757.872838

[17] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 81–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315460

[18] 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.

[19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254766

[20] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2013.01.010

[21] 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.

[22] 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. [Online]. Available: http://doi.acm.org/10.1145/2187671.2187677

[23] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 967–980. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376712

[24] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 330–339, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920886

[25] "Apache Avro," https://avro.apache.org/.

[26] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-013-0335-9

[27] 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. [Online]. Available: http://doi.acm.org/10.1145/2479942.2479946

[28] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740

[29] B. Cagri and T. Nesime, "Scalable data partitioning techniques for parallel sliding window processing over data streams," in *8th International Workshop on Data Management for Sensor Networks*, 2011. [Online]. Available: https://www.inf.ethz.ch/personal/cagri.balkesen/publications/dmsn2011.pdf

[30] 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. [Online]. Available: http://dx.doi.org/10.14778/2732240.2732245

[31] S. Chandrasekaran and M. J. Franklin, "Streaming queries over streaming data," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 203–214. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287388

[32] C. Mitch, B. Hari, B. Magdalena, C. Donald, C. Ugur, X. Ying, and Z. Stan, "Scalable distributed stream processing," in *First Biennial Conference on Innovative Data Systems Research*, 2003. [Online]. Available: http://cs.brown.edu/research/aurora/cidr03.pdf

[33] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 827–838. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007662

[34] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 311–322. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066193

[35] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: Stream processing under qos constraints at scale," *Cluster Computing*, vol. 17, no. 1, pp. 61–78, Mar. 2014.

[36] "Apache Kafka." https://kafka.apache.org/.

[37] K. Jay, N. Neha, and R. Jun, "Kafka: A distributed messaging system for log processing," in *Proceedings of 6th International Workshop on Networking Meets Databases*, ser. NetDB'11, 2011.

[38] "Apache ActiveMQ." http://activemq.apache.org/.

[39] "Rabbitmq," https://www.rabbitmq.com/.

[40] "Amqp," https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf.

[41] "Zeromq," http://zeromq.org/.

[42] "Hornetq," http://hornetq.jboss.org/.

[43] "Apache Flume." http://flume.apache.org/.

[44] L. Qiao, Y. Li, S. Takiar, Z. Liu, N. Veeramreddy, M. Tu, Y. Dai, I. Buenrostro, K. Surlaker, S. Das, and C. Botev, "Gobblin: Unifying data ingestion for hadoop," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1764–1769, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824073

[45] "Gobblin Documentation." https://gobblin.readthedocs.io/en/latest/.

[46] "Elasticsearch," https://www.elastic.co/products/elasticsearch.

[47] "Apache Sqoop." http://sqoop.apache.org/.

[48] 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," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1007/s00778-003-0095-z

[49] "Redis," https://redis.io/.

[50] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2926965

[51] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[52] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "Slik: Scalable low-latency indexes for a key-value store," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 57–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3026966

[53] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616488

[54] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 476–488. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750416

[55] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342360

[56] "Memcached," https://memcached.org/.

[57] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 385–398. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482663

[58] "Rocksdb," http://rocksdb.org/.

[59] "Lmdb," https://github.com/lmdbjava/lmdbjava.

[60] K. Florian and S. Michael, "Dxram: A persistent in-memory storage for billions of small objects," Dec. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6904240/

[61] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595631

[62] "Druid," http://druid.io/.

[63] S. Zoe and M. Kostas, "Scalable storage support for data stream processing," in *26th Symposium on Mass Storage Systems and Technologies*, ser. MSST'10. IEEE, 2010.

[64] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Rec.*, vol. 30, no. 3, pp. 109–120, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/603867.603884

[65] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ser. ALS'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268379.1268407

[66] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N. Tatbul, "Flexible and scalable storage management for data-intensive stream processing," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09. New York, NY, USA: ACM, 2009, pp. 934–945. [Online]. Available: http://doi.acm.org/10.1145/1516360.1516467

[67] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 480–491. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316732

[68] F. Dehne, D. E. Robillard, A. Rau-Chaplin, and N. Burke, "VOLAP: A scalable distributed system for real-time OLAP with high velocity data," in *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, 2016, pp. 354–363. [Online]. Available: http://dx.doi.org/10.1109/CLUSTER.2016.29

[69] D. Hilley and U. Ramachandran, "Persistent temporal streams," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09. New York, NY, USA: Springer-Verlag New York, Inc., 2009, pp. 17:1–17:20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1656980.1657003

[70] R. C. Fernandez, P. R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang, "Liquid: Unifying nearline and offline big data integration," in *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015. [Online]. Available: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf

[71] "Apache Samza." http://samza.apache.org/.

[72] W. Song, T. Gkountouvas, K. Birman, Q. Chen, and Z. Xiao, "The freeze-frame file system," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 307–320. [Online]. Available: http://doi.acm.org/10.1145/2987550.2987578

[73] "Timescaledb," http://www.timescaledb.com/.

[74] "Influxdb," https://github.com/influxdata/influxdb/.

[75] "Riak TS," http://basho.com/products/riak-ts/.

[76] "Opentsdb," http://opentsdb.net/.

[77] "Apache Kudu." https://kudu.apache.org/.

[78] "Apache HBase." https://hbase.apache.org/.

[79] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[80] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 199–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=2591305.2591325

[81] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[82] "Apache Cassandra." http://cassandra.apache.org/.

[83] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of the International Workshop on Memory Management*, ser. IWMM '95. London, UK, UK: Springer-Verlag, 1995, pp. 1–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=645647.664690

[84] "Carbondata," http://carbondata.incubator.apache.org/.

[85] "Apache Parquet," http://parquet.apache.org/.

[86] "Apache Orc," https://orc.apache.org/.

[87] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. on Knowl. and Data Eng.*, vol. 27, no. 7, 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7097722/

[88] 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. [Online]. Available: http://dx.doi.org/10.14778/1454159.1454179

[89] D. M., K. P., M. G., and T. M., "State access patterns in stream parallel computations," in *International Journal of High Performance Computing Applications (IJHPCA)*, 2017. [Online]. Available: http://pages.di.unipi.it/mencagli/publications/preprint-ijhpca-2017.pdf

[90] "Streaming-data algorithms for high-quality clustering," in *Proceedings of the 18th International Conference on Data Engineering*, ser. ICDE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 685–. [Online]. Available: http://dl.acm.org/citation.cfm?id=876875.878995

[91] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. d. Carvalho, and J. a. Gama, "Data stream clustering: A survey," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 13:1–13:31, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2522968.2522981

[92] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Rec.*, vol. 43, no. 1, pp. 9–20, May 2014. [Online]. Available: http://doi.acm.org/10.1145/2627692.2627694

[93] G. Sijie, D. Robin, and S. Leigh, "Distributedlog: A high performance replicated log service," in *IEEE 33rd International Conference on Data Engineering*, ser. ICDE'17. IEEE, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7930058/

[94] M. John, A. Cansu, Z. Stan, T. Nesime, and D. Jiang, "Data ingestion for the connected world," in *CIDR, Online Proceedings*, 2017.

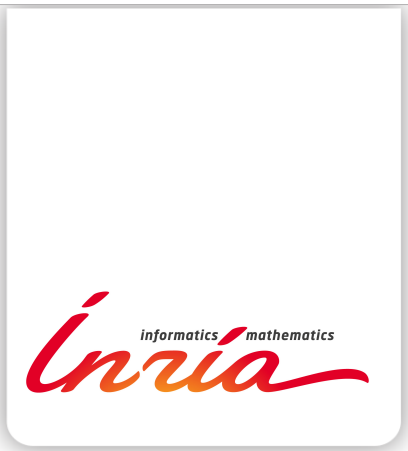
[95] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[96] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "Hopsfs: Scaling hierarchical file system metadata using newsql databases," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 89–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=3129633.3129642

[97] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane, "An optimized approach for storing and accessing small files on cloud storage," *J. Netw. Comput. Appl.*, vol. 35, no. 6, pp. 1847–1862, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2012.07.009

[98] S. Yang, "Iot stream processing and analytics in the fog," 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8004149/

[99] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 32:1–32:43, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3057266



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803