Special Issue

Wolfram Wingerath*, Felix Gessert, Steffen Friedrich, and Norbert Ritter

Real-time stream processing for Big Data

DOI 10.1515/itit-2016-0002
Received January 15, 2016; accepted May 2, 2016

Abstract: With the rise of the web 2.0 and the Internet of things, it has become feasible to track all kinds of information over time, in particular fine-grained user activities and sensor data on their environment and even their biometrics. However, while efficiency remains mandatory for any application trying to cope with huge amounts of data, only part of the potential of today's Big Data repositories can be exploited using traditional batch-oriented approaches as the value of data often decays quickly and high latency becomes unacceptable in some applications. In the last couple of years, several distributed data processing systems have emerged that deviate from the batch-oriented approach and tackle data items as they arrive, thus acknowledging the growing importance of timeliness and velocity in Big Data analytics.
In this article, we give an overview over the state of the art of stream processors for low-latency Big Data analytics and conduct a qualitative comparison of the most popular contenders, namely Storm and its abstraction layer Trident, Samza and Spark Streaming. We describe their respective underlying rationales, the guarantees they provide and discuss the trade-offs that come with selecting one of them for a particular task.

Keywords: Distributed real-time stream processing, Big Data analytics.

ACM CCS: General and reference → Document types → Surveys and overviews , Computer systems organization → Architectures → Distributed architectures → Cloud computing , Computer systems organization → Real-time systems → Real-time system architecture , Information systems → Data management systems → Database management system engines → Stream management , Computing methodologies → Distributed computing methodologies

***Corresponding author: Wolfram Wingerath,** Univ. of Hamburg, CS Dept., D-22527 Hamburg, Germany,
e-mail: wingerath@informatik.uni-hamburg.de
Felix Gessert, Steffen Friedrich, Norbert Ritter: Univ. of Hamburg, CS Dept., D-22527 Hamburg, Germany

1 Introduction

Through technological advance and increasing connectivity between people and devices, the amount of data available to (web) companies, governments and other organisations is constantly growing. The shift towards more dynamic and user-generated content in the web and the omnipresence of smart phones, wearables and other mobile devices, in particular, have led to an abundance of information that are only valuable for a short time and therefore have to be processed immediately. Companies like Amazon and Netflix have already adapted and are monitoring user activity to optimise product or video recommendations for the current user context. Twitter performs continuous sentiment analysis to inform users on trending topics as they come up and even Google has parted with batch processing for indexing the web to minimise the latency by which it reflects new and updated sites [34].

However, the idea of processing data in motion is not new: Complex Event Processing (CEP) engines [11, 12] and DBMSs with continuous query capabilities [36] can provide processing latency on the order of milliseconds and usually expose high-level, SQL-like interfaces and sophisticated querying functionalities like joins. But while typical deployments of these systems do not span more than a few nodes, the systems focused in this article have been designed specifically for deployments with 10s or 100s of nodes. Much like MapReduce, the main achievement of these new systems is abstraction from scaling issues and thus making development, deployment and maintenance of *highly scalable* systems feasible.

In the following, we provide an overview over some of the most popular distributed stream processing systems currently available and highlight similarities, differences and trade-offs taken in their respective designs.

Section 2 covers the environment in which the processing systems featured in this article are typically deployed, while the systems we focus on are described in Section 3. We give an overview over other systems for stream processing in Section 4 and conclude in Section 5.

2 Real-time analytics: Big Data in motion

In contrast to traditional data analytics systems that collect and periodically process huge – static – volumes of data, streaming analytics systems avoid putting data at rest and process it as it becomes available, thus minimising the time a single data item spends in the processing pipeline. Systems that routinely achieve latencies of several seconds or even subsecond latency between receiving data and producing output are often described as "real-time". However, large parts of today's Big Data infrastructure are built from distributed components that communicate via asynchronous network and are engineered on top of the JVM (Java Virtual Machine). Thus, these systems are only soft-real-time systems and never provide strict upper bounds on the time they take to produce an output.

Figure 1 illustrates typical layers of a streaming analytics pipeline. Data like user clicks, billing information or unstructured content such as images or text messages are collected from various places inside an organisation and then moved to the streaming layer (e.g. a queueing system like Kafka, Kinesis or ZeroMQ) from which it is accessible to a stream processor that performs a certain task to produce an output. This output is then forwarded to the serving layer which might for example be an analytics web GUI like trending topics at Twitter or a database where a materialised view is maintained.

In an attempt to combine the best of both worlds, an architectural pattern called the Lambda Architecture [32] has become quite popular that complements the slow batch-oriented processing with an additional real-time component and thus targets both the *Volume* and the *Velocity* challenge of Big Data [29] at the same time. As illustrated in Figure 2a, the Lambda Architecture describes a system comprising three layers: Data is stored

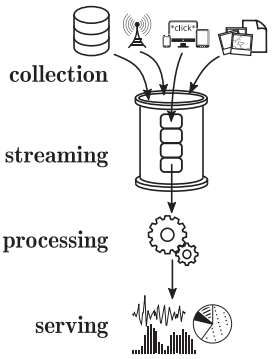
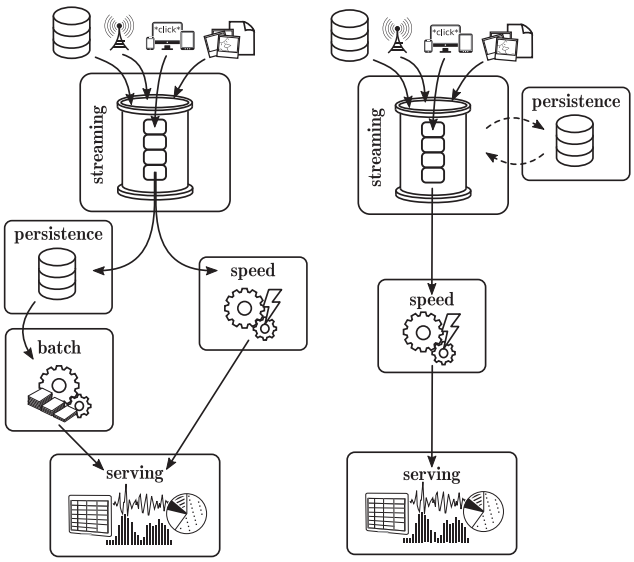


Figure 1: An abstract view on a streaming analytics pipeline.



(a) The Lambda Architecture. (b) The Kappa Architecture.

Figure 2: Lambda and Kappa Architecture in comparison.

in a persistence layer like HDFS from which it is ingested and processed by the batch layer periodically (e.g. once a day), while the speed layer handles the portion of the data that has not-yet been processed by the batch layer, and the serving layer consolidates both by merging the output of the batch and the speed layer. The obvious benefit of having a real-time system compensate for the high latency of batch processing is paid for by increased complexity in development, deployment and maintenance. If the batch layer is implemented with a system that supports both batch and stream processing (e.g. Spark), the speed layer often can be implemented with minimal overhead by using the corresponding streaming API (e.g. Spark Streaming) to make use of existing business logic and the existing deployment. For Hadoop-based and other systems that do not provide a streaming API, however, the speed layer is only available as a separate system. Using an abstract language like Summingbird [18] to write the business logic enables automatic compilation of code for both the batch and the stream processing system (e.g. Hadoop and Storm) and thus eases development in those cases where batch and speed layer can use (parts of) the same business logic, but the overhead for deployment and maintenance still remains.

Another approach that, in contrast, dispenses with the batch layer in favour of simplicity is known as the Kappa Architecture [25] and is illustrated in Figure 2b. The basic idea is to not periodically recompute all data in the batch layer, but to do all computation in the stream processing system alone and only perform recomputation

when the business logic changes by replaying historical data. To achieve this, the Kappa Architecture employs a powerful stream processor capable of coping with data at a far greater rate than it is incoming and a scalable streaming system for data retention. An example of such a streaming system is Kafka which has been specifically designed to work with the stream processor Samza in this kind of architecture. Archiving data (e.g. in HDFS) is still possible, but not part of the critical path and often not required as Kafka, for instance, supports retention times in the order of weeks. On the downside, however, the effort required to replay the entire history increases linearly with data volume and the naive approach of retaining the entire change stream may have significantly greater storage requirements then periodically processing the new data and updating an existing database, depending on whether and how efficiently the data is compacted in the streaming layer. As a consequence, the Kappa Architecture should only be considered an alternative to the Lambda Architecture in applications that do not require unbounded retention times or allow for efficient compaction (e.g. because it is reasonable to only keep the most recent value for each given key).

Of course, the latency displayed by the stream processor (speed layer) alone is only a fraction of the end-to-end application latency due to the impact of the network or other systems in the pipeline. But it is obviously an important factor and may dictate which system to choose in applications with strict timing SLAs. This article concentrates on the available systems for the stream processing layer.

3 Real-time processors

While all stream processors share some common ground regarding their underlying concepts and working principle, an important distinction between the individual systems that directly translates to the achievable speed of processing, i.e. latency, is the processing model as illustrated in Figure 3: Handling data items immediately as they arrive minimises latency at the cost of high per-item overhead (e.g. through messaging), whereas buffering and processing them in batches yields increased efficiency, but obviously increases the time the individual item spends in the data pipeline. Purely stream-oriented systems such as Storm and Samza provide very low latency and relatively high per-item cost, while batch-oriented systems achieve unparalleled resource-efficiency at the expense of latency that is prohibitively high for real-time applications.

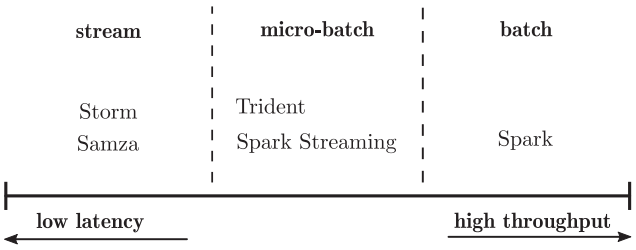


Figure 3: Choosing a processing model means trading off between latency and throughput.

The space between these two extremes is vast and some systems like Storm Trident and Spark Streaming employ micro-batching strategies to trade latency against throughput: Trident groups tuples into batches to relax the one-at-a-time processing model in favour of increased throughput, whereas Spark Streaming restricts batch size in a native batch processor to reduce latency. In the following, we go into more detail on the specificities of the above-mentioned systems and highlight inherent trade-offs and design decisions.

3.1 Storm

Storm (current stable version: 1.0.0) has been in development since late 2010, was open-sourced in September 2011 by Twitter and eventually became an Apache top-level project in 2014. It is the first distributed stream processing system to gain traction throughout research and practice and was initially promoted as the "Hadoop of real-time" [30, 31], because its programming model provided an abstraction for stream-processing similar to the abstraction that the MapReduce paradigm provides for batch-processing. But apart from being the first of its kind, Storm also has a wide user-base due to its compatibility with virtually any language: On top of the Java API, Storm is also Thrift-compatible and comes with adapters for numerous languages such as Perl, Python and Ruby. Storm can run on top of Mesos [5], as a dedicated cluster or even on a single machine. The vital parts of a Storm deployment are a ZooKeeper cluster for reliable coordination, several supervisors for execution and a Nimbus server to distribute code across the cluster and take action in case of worker failure; in order to shield against a failing Nimbus server, Storm allows having several hot-standby Nimbus instances. Storm is scalable, fault-tolerant and even elastic as work may be reassigned during runtime. As of version 1.0.0, Storm provides reliable state implementations that survive and recover from supervisor failure. Earlier versions focused on stateless processing and

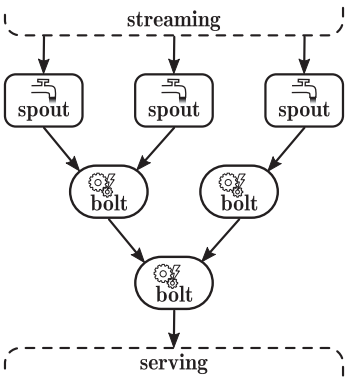


Figure 4: Data flow in a Storm topology: Data is ingested from the streaming layer and then passed between Storm components, until the final output reaches the serving layer.

thus required state management at the application level to achieve fault-tolerance and elasticity in stateful applications. Storm excels at speed and thus is able to perform in the realm of single-digit milliseconds in certain applications. Through the impact of network latency and garbage collection, however, real-world topologies usually do not display end-to-end latency below 50 ms [23, Chapter 7].

A data pipeline or application in Storm is called a topology and as illustrated in Figure 4 is a directed graph that represents data flow as directed edges between nodes which again represent the individual processing steps: The nodes that ingest data and thus initiate the data flow in the topology are called spouts and emit tuples to the nodes downstream which are called bolts and do processing, write data to external storage and may send tuples further downstream themselves. Storm comes with several groupings that control data flow between nodes, e.g. for shuffling or hash-partitioning a stream of tuples by some attribute value, but also allows arbitrary custom groupings. By default, Storm distributes spouts and bolts across the nodes in the cluster in a round-robin fashion, though the scheduler is pluggable to account for scenarios in which a certain processing step has to be executed on a particular node, for example because of hardware dependencies. The application logic is encapsulated in a manual definition of data flow and the spouts and bolts which implement interfaces to define their behaviour during start-up and on data ingestion or receiving a tuple, respectively.

While Storm does not provide any guarantee on the order in which tuples are processed, it does provide the option of at-least-once processing through an acknowledgement feature that tracks the processing status of every single tuple on its way through the topology: Storm will replay a tuple, if any bolt involved in processing it explicitly signals failure or does not acknowledge successful processing within a given timeframe. Using an appropriate streaming system, it is even possible to shield against spout failures, but the acknowledgement feature is often not used at all in practice, because the messaging overhead imposed by tracking tuple lineage, i.e. a tuple and all the tuples that are emitted on its behalf, noticeably impairs achievable system throughput [20]. With version 1.0.0, Storm introduced a backpressure mechanism to throttle data ingestion as a last resort whenever data is ingested faster than it can be processed. If processing becomes a bottleneck in a topology without such a mechanism, throughput degrades as tuples eventually time-out and are either lost (at-most-once processing) or replayed repeatedly to possibly time-out again (at-least-once processing), thus putting even more load on an already overburdened system.

3.1.1 Storm Trident

In autumn 2012 and version 0.8.0, Trident was released as a high-level API with stronger ordering guarantees and a more abstract programming interface with built-in support for joins, aggregations, grouping, functions and filters. In contrast to Storm, Trident topologies are directed *acyclic* graphs (DAGs) as they do not support cycles; this makes them less suitable for implementing iterative algorithms and is also a difference to plain Storm topologies which are often wrongfully described as DAGs, but actually can introduce cycles. Also, Trident does not work on individual tuples, but on micro-batches and correspondingly introduces batch size as a parameter to increase throughput at the cost of latency which, however, may still be as low as several milliseconds for small batches [22]. All batches are by default processed in sequential order, one after another, although Trident can also be configured to process multiple batches in parallel. On top of Storm's scalability and elasticity, Trident provides its own API for fault-tolerant state management with exactly-once processing semantics. In more detail, Trident prevents data loss by using Storm's acknowledgement feature and guarantees that every tuple is reflected only once in persistent state by maintaining additional information alongside state and by applying updates transactionally. As of writing, two variants of state management are available: One only stores the sequence number of the last-processed batch together with current state, but may block the entire topology when one or more tuples of a failed batch cannot be replayed (e.g. due to unavailability of the data source), whereas the other can tolerate this kind of failure, but is more heavyweight as it also stores the last-known state.

Irrespective of whether batches are processed in parallel or one by one, state updates have to be persisted in strict order to guarantee correct semantics. As a consequence, their size and frequency can become a bottleneck and Trident can therefore only feasibly manage small state.

3.2 Samza

Samza (current stable version: 0.10.0) is very similar to Storm in that it is a stream processor with a one-at-a-time processing model and at-least-once processing semantics. It was initially created at LinkedIn, submitted to the Apache Incubator in July 2013 and was granted top-level status in 2015. Samza was co-developed with the queueing system Kafka [4, 26] and therefore relies on the same messaging semantics: Streams are partitioned and messages (i.e. data items) inside the same partition are ordered, whereas there is no order between messages of different partitions. Even though Samza can work with other queueing systems, Kafka's capabilities are effectively required to use Samza to its full potential and therefore it is assumed to be deployed with Samza for the rest of this section. In comparison to Storm, Samza requires a little more work to deploy as it does not only depend on a ZooKeeper cluster, but also runs on top of Hadoop YARN [6] for fault-tolerance: In essence, application logic is implemented as a job that is submitted through the Samza YARN client which has YARN then start and supervise one or more containers. Scalability is achieved through running a Samza job in several parallel tasks each of which consumes a separate Kafka partition; the degree of parallelism, i.e. the number of tasks, cannot be increased dynamically at runtime. Similar to Kafka, Samza focuses on support for JVM-languages, particularly Java. Contrasting Storm and Trident, Samza is designed to handle *large amounts of state* in a fault-tolerant fashion by persisting state in a local database and replicating state updates to Kafka. By default, Samza employs a key-value store for this purpose, but other storage engines with richer querying capabilities can be plugged in.

As illustrated in Figure 5, a Samza job represents one processing step in an analytics pipeline and thus roughly corresponds to a bolt in a Storm topology. In stark contrast to Storm, however, where data is directly sent from one bolt to another, output produced by a Samza job is always written back to Kafka from where it can be consumed by other Samza jobs. Although a single Samza job or a single Kafka persistence hop may delay a message by only a few milliseconds [8, 24], latency adds up and complex analytics pipelines comprising several processing steps eventually display higher end-to-end latency than comparable Storm implementations.

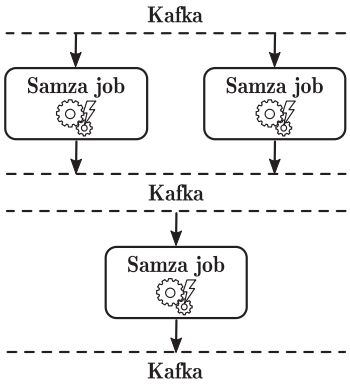


Figure 5: Data flow in a typical Samza analytics pipeline: Samza jobs cannot communicate directly, but have to use a queueing system such as Kafka as message broker.

However, this design also decouples individual processing steps and thus eases development. Another advantage is that buffering data between processing steps makes (intermediate) results available to unrelated parties, e.g. other teams in the same company, and further eliminates the need for a backpressure algorithm, since there is no harm in the backlog of a particular job filling up temporarily, given a reasonably sized Kafka deployment. Since Samza processes messages in order and stores processing results durable after each step, it is able to prevent data loss by periodically checkpointing current progress and reprocessing all data from that point onwards in case of failure; in fact, Samza does not support a weaker guarantee than at-least-once processing, since there would be virtually no performance gain in relaxing this guarantee. While Samza does not provide exactly-once semantics, it allows configuring the checkpointing interval and thus offers some control over the amount of data that may be processed multiple times in an error scenario.

3.3 Spark streaming

The Spark framework [39] (current stable version: 1.6.1) is a batch-processing framework that is often mentioned as the inofficial successor of Hadoop as it offers several benefits in comparison, most notably a more concise API resulting in less verbose application logic and significant performance improvements through in-memory caching: In particular, iterative algorithms (e.g. machine learning algorithms such as k-means clustering or logistic regression) are accelerated by orders of magnitude, because data is not necessarily written to and loaded from disk between

every processing step. In addition to these performance benefits, Spark provides a variety of machine learning algorithms out-of-the-box through the MLlib library. Originating from UC Berkeley in 2009, Spark was open-sourced in 2010 and was donated to the Apache Software Foundation in 2013 where it became a top-level project in February 2014. It is mostly written in Scala and has a Java, Scala and Python API. The core abstraction of Spark are distributed and immutable collections called RDDs (resilient distributed datasets) that can only be manipulated through deterministic operations. Spark is resilient to machine failures by keeping track of any RDD's lineage, i.e. the sequence of operations that created it, and checkpointing RDDs that are expensive to recompute, e.g. to HDFS [35]. A Spark deployment consists of a cluster manager for resource management (and supervision), a driver program for application scheduling and several worker nodes to execute the application logic. Spark runs on top of Mesos, YARN or in standalone mode in which case it may be used in combination with ZooKeeper to remove the master node, i.e. the cluster manager, as a single point of failure.

Spark Streaming [40] shifts Spark's batch-processing approach towards real-time requirements by chunking the stream of incoming data items into small batches, transforming them into RDDs and processing them as usual. It further takes care of data flow and distribution automatically. Spark Streaming has been in development since late 2011 and became part of Spark in February 2013. Being a part of the Spark framework, Spark Streaming had a large developer community and also a huge group of potential users from day one, since both systems share the same API and since Spark Streaming runs on top of a common Spark cluster. Thus, it can be made resilient to failure of any component [37] like Storm and Samza and further supports dynamically scaling the resources allocated for an application. Data is ingested and transformed into a sequence of RDDs which is called DStream (discretised stream) before processing through workers. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees. Since there is a certain job scheduling delay when processing an RDD, batch sizes below 50 ms tend to be infeasible [10, section "Performance Tuning"]. Accordingly, processing an RDD takes around 100 ms in the best case, although Spark Streaming is designed for latency in the order of a few seconds [40, Sec. 2]. To prevent data loss even for unreliable data sources, Spark Streaming grants the option of using a write-ahead log (WAL) from which data can be replayed after failure. State management is realised through a state DStream that can be updated through a DStream transformation.

3.4 Discussion

Table 1 sums up the properties of these systems in direct comparison. Storm provides low latency, but does not offer ordering guarantees and is often deployed providing no delivery guarantees at all, since the per-tuple acknowledgement required for at-least-once processing effectively doubles messaging overhead. Stateful exactly-once processing is available in Trident through idempotent state updates, but has notable impact on performance and even fault-tolerance in some failure scenarios. Samza is another native stream processor that has not been geared towards low latency as much as Storm and puts more focus on providing rich semantics, in particular through a built-in concept of state management. Having been developed for use with Kafka in the Kappa Architecture, Samza and Kafka are tightly integrated and share messaging semantics; thus, Samza can fully exploit the ordering guarantees provided by Kafka. Spark Streaming effectively unifies batch and stream processing and offers a high-level API, exactly-once processing guarantees and a rich set of libraries, all of which can greatly reduce the complexity of application development. However, being a native batch

Table 1: Apache Storm/Trident, LinkedIn's Samza and Apache Spark Streaming in direct comparison.

	Storm	Trident	Samza	Spark Streaming
strictest guarantee	at-least-once	exactly-once	at-least-once	exactly-once
achievable latency	$\ll 100$ ms	< 100 ms	< 100 ms	< 1 s
state management	yes	yes (small state)	yes	yes
processing model	one-at-a-time	micro-batch	one-at-a-time	micro-batch
backpressure mechanism	yes	yes	not required (buffering)	yes
ordering guarantees	no	between batches	within stream partitions	between batches
elasticity	yes	yes	no	yes

processor, Spark Streaming loses to its contenders with respect to latency [20].

All these different systems show that low latency is involved in a number of trade-offs with other desirable properties such as throughput, fault-tolerance, reliability (processing guarantees) and ease of development. Throughput can be optimised by buffering data and processing it in batches to reduce the impact of messaging and other overhead per data item, whereas this obviously increases the in-flight time of individual data items. Abstract interfaces hide system complexity and ease the process of application development, but sometimes also limit the possibilities of performance tuning. Similarly, rich processing guarantees and fault-tolerance for stateful operations increase reliability and make it easier to reason about semantics, but require the system to do additional work, e.g. acknowledgements and (synchronous) state replication. Exactly-once semantics are particularly desirable and can be implemented through combining at-least-once guarantees with either transactional or idempotent state updates, but they cannot be achieved for actions with side-effects such as sending a notification to an administrator.

4 Further systems

In the last couple of years, a great number of stream processors have emerged that all aim to provide high availability, fault-tolerance and horizontal scalability. Flink [2] (formerly known as Stratosphere [15]) is a project that has many parallels to Spark Streaming as it also originated from research and advertises the unification of batch and stream processing in the same system, providing exactly-once guarantees for the stream programming model and a high-level API comparable to that of Trident. In contrast to Spark Streaming, though, Flink does not rely on batching for stream processing internally and thus delivers low latency in the order of Storm or Samza. Project Apex [1] is the open-sourced DataTorrent RTS core engine. Much like Flink, Apex promises high performance in stream and batch processing with low latency in streaming workloads. As Spark/Spark Streaming, Flink and Apex are complemented by a host of database, file system and other connectors as well as pattern matching, machine learning and other algorithms through additional libraries. A system that is not publicly available is Twitter's Heron [27]. Designed to replace Storm at twitter, Heron is completely API-compatible to Storm, but improves on several aspects such as backpressure, efficiency, resource isolation, multitenancy, ease of debugging and performance monitoring, but reportedly does not provide exactly-once delivery guarantees. MillWheel [13] is an extremely scalable stream processor that offers similar qualities as Flink and Apex, e.g. state management and exactly-once semantics. Millwheel and FlumeJava [19] are the execution engines behind Google's Dataflow cloud service for data processing. Like other Google services and unlike most other systems discussed in this section, Dataflow is fully managed and thus relieves its users of the burden of deployment and all related troubles. The Dataflow programming model [14] combines batch and stream processing and is also agnostic of the underlying processing system, thus decoupling business logic from the actual implementation. The runtime-agnostic API was open-sourced in 2015 and has evolved into the Apache Beam project (short for Batch and stream, currently incubating) [7] to bundle it with the corresponding execution engines (runners): As of writing, Flink, Spark and the proprietary Google Dataflow cloud service are supported. The only other fully managed stream processing system apart from Google Dataflow that we are aware of is IBM Infosphere Streams [17]. However, in contrast to Google Dataflow which is documented to be highly scalable (quota limit for customers: 1000 compute nodes [9]), it is hard to find evidence for high scalability of IBM Infosphere Streams; performance evaluations made by IBM [21] only indicate it performs well in small deployments with up to a few nodes. Apache Flume [3] is a system for efficient data aggregation and collection that is often used for data ingestion into Hadoop as it integrates well with HDFS and can handle large volumes of incoming data. But Flume also supports simple operations such as filtering or modifying on incoming data through Flume Interceptors [23, Chapter 7] which may be chained together to form a low-latency processing pipeline.

The list of distributed stream processors goes on [16, 28, 33, 38], but since a complete discussion of the Big Data stream processing landscape is out of the scope of this article, we do not go into further detail here.

5 Wrap-up

Batch-oriented systems have done the heavy lifting in data-intensive applications for decades, but they do not reflect the unbounded and continuous nature of data as it is produced in many real-world applications. Stream-oriented systems, on the other hand, process data as it arrives and thus are oftentimes a more natural fit, though inferior with respect to efficiency. While a growing number of production deployments implementing the Lambda Architecture and emerging hybrid systems like Dataflow/Beam, Flink or Apex document significant efforts to close the

gap between batch and stream-processing in both research and practice, the Kappa Architecture completely eschews the traditional approach and even heralds the advent of purely stream-oriented Big Data analytics. However, whether at the core of novel system designs or as a complement to existing architectures, horizontally scalable stream processors are gaining momentum as the requirement for low latency has become a driving force in modern Big Data analytics pipelines.

References

1. Apache apex. http://apex.incubator.apache.org/.
2. Apache flink. https://flink.apache.org/.
3. Apache Flume. https://flume.apache.org/.
4. Apache Kafka. http://kafka.apache.org/.
5. Apache Mesos. http://mesos.apache.org/.
6. Apache Yarn. http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.
7. Beamproposal – incubator wiki. https://wiki.apache.org/incubator/BeamProposal.
8. Comparisons: Spark streaming. http://samza.apache.org/learn/documentation/0.7.0/comparisons/spark-streaming.html. Accessed: 2016-01-12.
9. Resource quotas. https://cloud.google.com/dataflow/quotas. Accessed: 2016-01-14.
10. Spark streaming programming guide. https://spark.apache.org/docs/1.6.0/streaming-programming-guide.html. Accessed: 2016-01-12.
11. D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.
12. 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*, 12(2):120–139, Aug. 2003.
13. T. Akidau, A. Balikov, K. Bekiroglu, et al. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.
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. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.
15. A. Alexandrov, R. Bergmann, S. Ewen, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 2014.
16. R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD '13*, 2013.
17. A. Biem, E. Bouillet, H. Feng, et al. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.
18. O. Boykin, S. Ritchie, I. O'Connell, et al. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB*, 2014.
19. C. Chambers, A. Raniwala, F. Perry, et al. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI 2010*, 2010.
20. S. Chintapalli, D. Dagit, B. Evans, et al. Benchmarking streaming computation engines at yahoo! http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, December 2015. Accessed: 2016-01-11.
21. I. Corporation. Of streams and storms. Technical report, IBM Software Group, 2014.
22. Ericsson. Trident – benchmarking performance. http://www.ericsson.com/research-blog/data-knowledge/trident-benchmarking-performance/. Accessed: 2016-01-12.
23. M. Grover, T. Malaska, J. Seidman, and G. Shapira. *Hadoop Application Architectures*. O'Reilly, Beijing, 2015.
24. J. Kreps. Benchmarking apache kafka: 2 million writes per second (on three cheap machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines. Accessed: 2016-01-12.
25. J. Kreps. Questioning the lambda architecture. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html, 7 2014. Accessed: 2015-12-17.
26. J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB'11*, 2011.
27. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.
28. W. Lam, L. Liu, S. Prasad, et al. Muppet: Mapreduce-style processing of fast data. *VLDB 2012*, 2012.
29. D. Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.
30. N. Marz. Preview of storm: The hadoop of realtime processing. http://web.archive.org/web/20120509023348/http://tech.backtype.com/preview-of-storm-the-hadoop-of-realtime-proce, 5 2012. Accessed: 2015-12-17.
31. N. Marz. History of apache storm and lessons learned. http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html, 10 2014. Accessed: 2015-12-17.
32. N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.
33. L. Neumeyer, B. Robbins, and A. Kesari. S4: Distributed stream computing platform. In *In Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud*, 2010.
34. D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
35. 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)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

36. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD Rec.*, 21(2):321–330, June 1992.
37. B. Venkat, P. Padmanabhan, A. Arokiasamy, and R. Uppalapati. Can spark streaming survive chaos monkey? http://techblog.netflix.com/2015/03/can-spark-streaming-survive-chaos-monkey.html, Marriage 2015. Accessed: 2016-01-11.
38. F. Yang, Z. Qian, X. Chen, I. Beschastnikh, L. Zhuang, L. Zhou, and G. Shen. Sonora: A platform for continuous mobile-cloud computing. Technical Report MSR-TR-2012-34, March 2012.
39. M. Zaharia, M. Chowdhury, T. Das, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
40. M. Zaharia, T. Das, H. Li, et al. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

Bionotes

Wolfram Wingerath
Univ. of Hamburg, CS Dept.,
D-22527 Hamburg, Germany
wingerath@informatik.uni-hamburg.de

Wolfram Wingerath is a Ph.D. student under supervision of Norbert Ritter teaching and researching at the University of Hamburg. He was co-organiser of the BTW 2015 conference and has held workshop and conference talks on his published work on several occasions. Wolfram is part of the databases and information systems group and his research interests evolve around scalable NoSQL database systems, cloud computing and Big Data analytics, but he also has a background in data quality and duplicate detection. His current work is related to real-time stream processing and explores the possibilities of providing always-up-to-date materialised views and continuous queries on top of existing non-streaming DBMSs.

Felix Gessert
Univ. of Hamburg, CS Dept.,
D-22527 Hamburg, Germany
gessert@informatik.uni-hamburg.de

Felix Gessert is a Ph.D. student at the databases and information systems group at the University of Hamburg. His main research fields are scalable database systems, transactions and web technologies for cloud data management. His thesis addresses caching and transaction processing for low-latency mobile and web applications. He is also founder and CEO of the startup Baqend that implements these research results in a cloud-based backend-as-a-service platform. Since their product is based on a polyglot, NoSQL-centric storage model, he is very interested in both the research and practical challenges of leveraging and improving these systems. He is frequently giving talks on different NoSQL topics.

Steffen Friedrich
Univ. of Hamburg, CS Dept.,
D-22527 Hamburg, Germany
friedrich@informatik.uni-hamburg.de

Steffen Friedrich is a Ph.D. student working under supervision of Norbert Ritter at the University of Hamburg. He has taken part in several workshops and conferences, both as presenter (e.g. DMC2014) and as co-organiser (BTW 2015). Being a member of the databases and information systems group, Steffen is interested in large-scale data management and data-intensive computing. Furthermore, in his Master thesis, he also dealt with data quality issues, specifically with duplicate detection in probabilistic data. His research project is primarily concerned with benchmarking of non-functional characteristics (e.g. consistency and availability) in distributed NoSQL database systems.

Prof. Dr.-Ing. Norbert Ritter
Univ. of Hamburg, CS Dept.,
D-22527 Hamburg, Germany
ritter@informatik.uni-hamburg.de

Prof. Dr.-Ing. Norbert Ritter is a full professor of computer science at the University of Hamburg, where he heads the databases and information systems group. He received his Ph.D. from the University of Kaiserslautern in 1997. His research interests include distributed and federated database systems, transaction processing, caching, cloud data management, information integration and autonomous database systems. He has been teaching NoSQL topics in various courses for several years. Seeing the many open challenges for NoSQL systems, he and Felix Gessert have been organizing the annual Scalable Cloud Data Management Workshop (www.scdm2015.com) for three years to promote research in this area.