Latest updates: https://dl.acm.org/doi/10.1145/2611286.2611309

RESEARCH-ARTICLE

Cloud-based data stream processing

THOMAS HEINZE, SAP SE, Walldorf, Baden-Wurttemberg, Germany
LEONARDO ANIELLO, Sapienza University of Rome, Rome, RM, Italy
LEONARDO QUERZONI, Sapienza University of Rome, Rome, RM, Italy
ZBIGNIEW JERZAK, SAP SE, Walldorf, Baden-Wurttemberg, Germany

Open Access Support provided by:
Sapienza University of Rome
SAP SE

PDF Download
2611286.2611309.pdf
13 February 2026
Total Citations: 43
Total Downloads: 998

Published: 26 May 2014

Citation in BibTeX format

DEBS '14: The 8th ACM International Conference on Distributed Event-Based Systems
May 26 - 29, 2014
Mumbai, India

Conference Sponsors:
SIGSOFT
SIGMOD

Petitioner has added a romanette to page (i).
Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1008

DEBS '14: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (May 2014)
https://doi.org/10.1145/2611286.2611309
ISBN: 9781450327374

Tutorial: Cloud-based Data Stream Processing

Thomas Heinze[1], Leonardo Aniello[2], Leonardo Querzoni[2], Zbigniew Jerzak[1]
[1]SAP AG [2]CIS - Sapienza University of Rome
{firstname.lastname}@sap.com {lastname}@diag.uniroma1.it

ABSTRACT

In this tutorial we present the results of recent research about the cloud enablement of data streaming systems. We illustrate, based on both industrial as well as academic prototypes, new emerging uses cases and research trends. Specifically, we focus on novel approaches for (1) scalability and (2) fault tolerance in large scale distributed streaming systems. In general, new fault tolerance mechanisms strive to be more robust and at the same time introduce less overhead. Novel load balancing approaches focus on elastic scaling over hundreds of instances based on the data and query workload. Finally, we present open challenges for the next generation of cloud-based data stream processing engines.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

Keywords

Cloud-based Data Stream Processing, Load Balancing, Fault Tolerance

1. INTRODUCTION

A data stream processing system executes a set of continous queries over a potentially unbounded data stream. Thereby, the system constantly outputs new results on the fly. Typical use cases include financial trading and monitoring of manufacturing equipment or logistics data. These scenarios require a high throughput and a low end to end latency from the system, despite possible fluctuations in the workload.

In the last decade, a large number of different academic prototypes as well as commercial products have been built to fulfill these requirements (see Figure 1). In general, data stream processing systems can be divided into 3 generations:

First generation stream processing systems have been built as stand-alone prototypes or as extensions of existing database engines. They were developed with a specific use case in mind and are very limited regarding the supported operator types as well as available functionalities. Representatives of this generation include Niagara [29], Telegraph [11] and Aurora [2].

Second generation systems extended the ideas of data stream processing with advanced features such as fault tolerance [1], adaptive query processing [34], as well as an enhanced operator expressiveness [7]. Important examples of this class are Borealis [1], CEDR [7], System S [22] and CAPE [34].

Third generation system design is strongly driven by the trend towards cloud computing, which requires the data stream processing engines to be highly scalable and robust towards faults. Well-known systems of this generation include Apache S4 [30], D-Streams [42], Storm [27] and StreamCloud [18].

In this tutorial we illustrate the research trends for the third generation of data stream processing engines. We describe the key requirements, which led to the development of this new generation of systems. In addition, we present detailed technical insights as well as future research directions.

The focus of this tutorial is on three key differentiators to previous research: (1) the intended use case, (2) the used scaling mechanisms and (3) the provided fault tolerance mechanisms. We present how the data stream processing system allows a user to scale out to hundreds of processing nodes, and ensures a fault tolerant execution even in an error-prone environment. We highlight key achievements in recent research and outline the difference to traditional data stream processing systems.

In the following, we first (Section 2) illustrate the use cases that drove the design of cloud-based data stream processing. Afterwards, we detail state of the art techniques for both scalability and fault-tolerance of a data stream processing engine in Section 3 and 4 respectively. Finally, we illustrate possible directions for future research in Section 5.

2. INTENDED USES CASES

In the third generation of data streaming systems a new class of application areas has been introduced. These include anomaly detection within social network data streams [3, 27, 30, 33] and website/infrastructure monitoring [30, 42]. All these use cases have a set of common requirements: (1) the scenarios typically involve input streams with high up to very high data rates (> 10000 event/s), (2) they have

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
DEBS'14, May 26-29, 2014, MUMBAI, India.
Copyright 2014 ACM 978-1-4503-2737-4/14/05 ...$15.00.
http://dx.doi.org/10.1145/2611286.2611309.

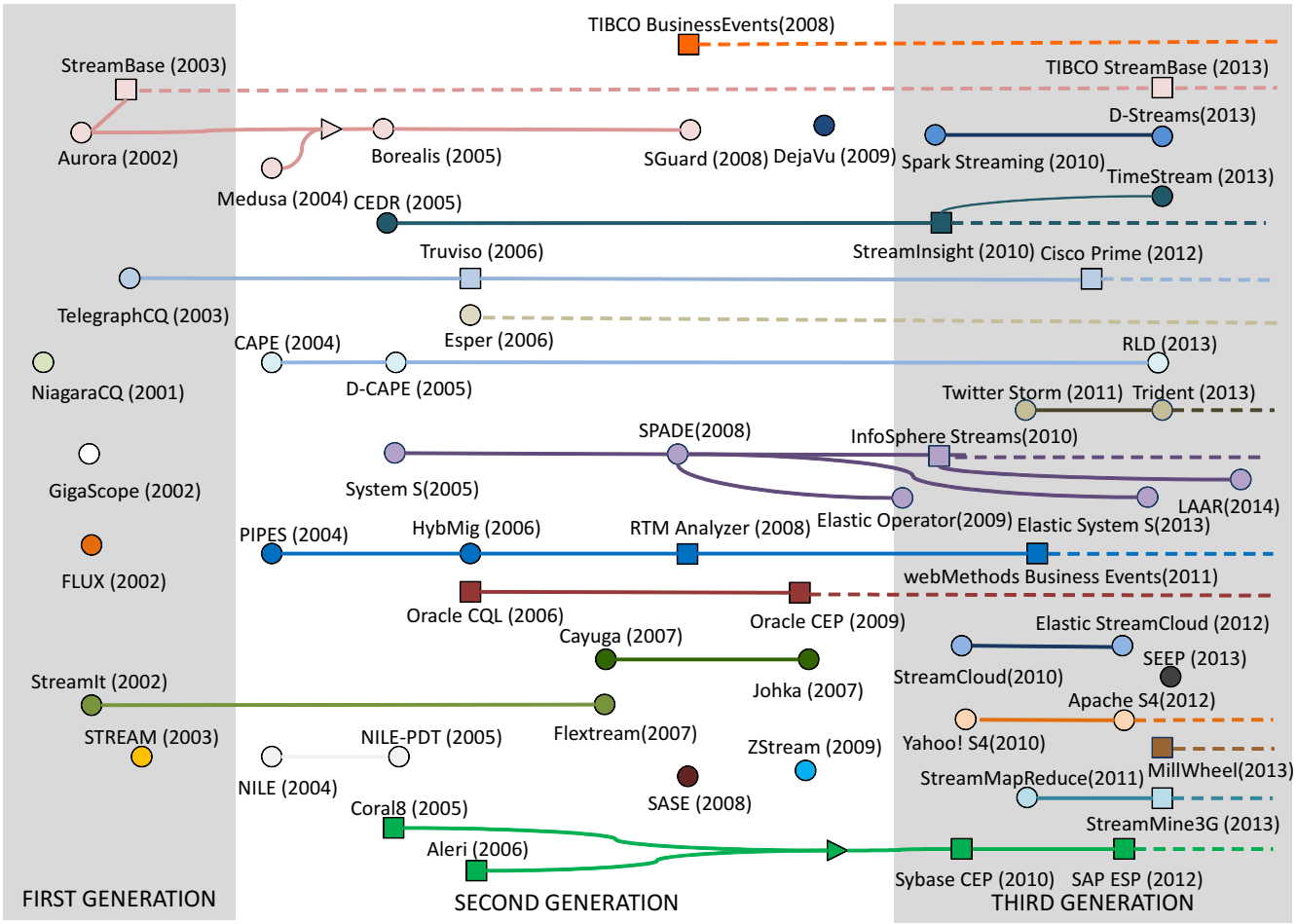

Figure 1: History of data stream processing systems

relaxed latency constraints (up to a few seconds), (3) the use cases require the correlation among historical and live data, (4) they require systems to elastically scale and to support diverse workloads and (5) they need low overhead fault tolerance supporting out of order events and exactly once semantic. In the following, we illustrate some of these use cases in detail.

Google Zeitgeist [3], which tracks web queries trend evolution at runtime, is an example of a data stream processing application characterized by the aforementioned requirements. Zeitgeist is designed to analyze queries from Google search (hence the very high data rates) to build a historical model for each query and then identify anomalies, like spiking or dipping searches as quickly as possible. Zeitgeist collects incoming search queries in 1-second buckets; buckets are then compared to historical data represented by known search query models. Zeitgeist is an application running on top of MillWheel [3], Google's data stream processing system.

Whenever a relevant event happens in the world, people use Twitter to discover what is happening by issuing queries. In such cases, significant spikes occur in the queries submitted to Twitter, and it is very likely that these queries have not been seen before, making it really challenging to correctly relate them to the events people are actually looking for. Twitter has to manage 2.1 billion queries per day[1], and needs to cope with such occasional spikes by correlating queries with tweets in real-time in order to provide results as accurate as possible. Furthermore, the interest of people for these relevant events is temporary and the corresponding query spikes fade away within a limited time window, so it is mandatory to sharpen this correlation as quickly as possible. At this aim, Twitter employs Storm [27] to spot popular queries so as to analyze them in details and achieve an improved accuracy in result generation [38].

Yahoo! personalizes search advertising on the basis of users' queries. In order to improve the accuracy of the models employed to predict advertisement relevance, this kind of analysis requires the evaluation in real-time of thousands of queries per second submitted by millions of distinct users. Yahoo! used S4 [30] to cope with this requirement.

Dublin City Council had the need to enhance the monitoring of its 1000 buses with the aim of delivering the best possible public transportation services to its 1.2 million citizens. They employ System S [17] to track the position of buses through GPS signals and to display real-time traffic information through dashboards. By continuously monitoring bus movements across the city, it is possible to accurately

[1] `http://www.statisticbrain.com/twitter-statistics/`

foresee arrival times and to suggest the best routes to take in case of traffic congestions or car accidents [21].

Aggregating online advertising flowing at a rate of 700K URLs per second, and executing sentiment analysis on 10K tweets per second, both within a 2-second maximum latency, are real cases where TimeStream [33] is used.

Fraud detection in cellular telephony is enforced by processing in real-time a number of Call Detail Records (CDRs) in the order of 10K-50K CDRs per second. The queries to be used to detect frauds include self-joins on the CDR stream that require the comparison of millions of CDR pairs per second. The challenges of this scenario have been addressed by employing StreamCloud [18].

3. SCALING MECHANISMS

An important aspect of any data stream processing system is its ability to scale with increasing load, where the load is characterized by the number and complexity of issued queries and the rate of incoming events to be analyzed. Cloud-based data streaming engines, in particular, are designed to dynamically scale to hundreds of computing nodes and automatically cope with varying workloads. The design of such engines poses two major technical challenges:

Data Partitioning The major challenge is to allow parallel evaluation of multiple data elements ensuring semantical correctness. This involves a reasonable data partitioning and merging scheme as well as mechanisms to detect points for parallelization.

Query Partitioning If a single host is not able to process all incoming data or all queries running in the system a distributed setup is applied. This involves the question, how to distribute the load across available hosts and how to achieve a load balance between these machines.

3.1 Data Partitioning

Early approaches for allowing parallel execution of data stream processing systems have been proposed with FLUX [37] and Aurora [2]. Both systems present a scheme for parallelizing operators by introducing a splitter and merge operator. Depending on the type of the parallelized operator, the merge operator can be either a simple union of all streams or might include also a sorting function [1] (See Figure 2). A simple filter can be parallelized using a round robin scheme where each instance can process any input event. The output of the available parallel instances needs only to be aggregated with a union. In contrast, for an aggregation operator all data elements pertaining to a same key group need to be processed by the same instance to ensure semantical correctness. In addition, the output of all instances needs to be sorted based on a timestamp to ensure temporal ordering.

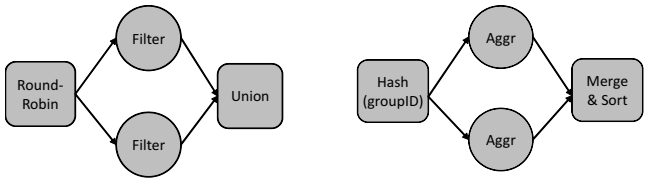

Figure 2: Examples for a parallel operator

.

Parallelization is a central feature of any cloud-based data stream processing system. Therefore, both Apache S4 [30] and Storm [27] allow the design of parallel applications. Within Storm a user can express data parallelism by defining the number of parallel tasks per operator. Apache S4 [30] creates for each new key in the data stream a *processing element*, which is than executed one of the running *processing node*. In both approaches the user needs to understand the data parallelism and explicitely enforce in its code the sequential ordering.

An alternative approach is taken by Hadoop Online [13] and StreamMapReduce [9], which presents a methods to implement stream-based tasks through the map-reduce programming paradigm. The authors extend the notion of *map* and *reduce* with a *stateful reducer* to overcome the strict phasing. The approach allows the usage of the MapReduce paradigm for streaming use cases, which allows a custom and highly parallelized execution.

An auto-parallelization approach has recently be proposed as extension of System S [36]. This system consists of a combination of compiler and runtime, which automatically enforces data parallelism at the operator level. The compiler detects regions for parallelization, while the system runtime guarantees that output tuples follow the same order as for a sequential execution. This approach supports classical data stream operators, custom operators as well as pattern matching operators [19].

Another research direction is based on elastic data parallelism. Elasticity describes the behavior of a system to dynamically scale in or out based on a changing workload [5]. The goal of an elastic scaling system is to react to unpredicted workload changes and minimize the monetary cost for the user. The notion of elasticity was first used within data stream processing systems by Schneider et al. for System S [35]. The system varies the number of threads used for pre-defined operator during runtime. If the load increases, additional threads are launched. Similarly if the load decreases, the number of running threads is reduced. Gedik et al. [17] combine the ideas of auto parallelization and elasticity within a single prototype, which leveraging a controller-based architecture, automatically determines the optimal number of partitions to be used within the system.

3.2 Query Partitioning

A fundamental problem for a distributed data stream processing system is to efficiently use the computational resources (hosts) available in the data center. The problem of assigning a set of operators to a set of available hosts is usually referred to as the *operator placement problem*. Many different placement algorithms have been presented for different use cases. An overview of alternative approaches is presented by Lakshmanan et al. [24]. The most recent algorithms have been presented by SODA [39], SQPR [23], MACE [10] and [4] for the Storm system.

All placement algorithms can be differentiated based on multiple factors, the most important ones include:

Optimization goal. Depending on the intended use case the placement algorithm tries to optimize different objectives to overcome the limitations of the current setup. These potential goals include a limited CPU bandwidth, a limited network bandwidth [31], latency optimization [10] or load balance within the system [39].

(i)

Execution model. The algorithms can be differentiated on the basis of the execution model they employ. A *centralized approach* [39, 23] collects all statistical information within a single manager and derives placement decisions for the whole network. In contrast, in a *decentralized approach* [31] decisions are taken as the result of a collaborative effort among all hosts. While the latter provides better scalability, a centralized manager can normally provide more optimized placement decisions as it can reason on global system knowledge.

Algorithm runtime. The decision can be either made offline based on an estimation of the workload [40], online based on an adaptive algorithm measuring the workload or based on a combination of both [23, 39].

An important aspect of the design of a distributed data stream processing system is the mechanism used to migrate operators among hosts. The migration of stateful operators, like aggregation or join operators, requires both the rewiring of data streams and the extraction and replay of the operator current state. The mechanisms used for this process are similar to those used to extract checkpoints or operator migration techniques used within adaptive query optimization systems [43, 41]. Based on the recent research two classes of operator movement protocols can be differentiated:

Pause & Resume. This approach [37] extracts the current state from the old instance and replays it within the new instance. Therefore, the operator needs to be paused to ensure a semantically correct migration. The drawback of this approach is that it creates an observable latency peak during the execution.

Parallel Track In the Parallel Track approach [18, 43] both the old and the new operator instances concurrently produce partial output. Thereby, the old instance produces still output until the state of both instances are in sync. Depending on the implementation the new instances either produce only partial results or no results during the synchronization time. This approach has the advantage of no latency peak during an operator movement. However, it requires enhanced mechanisms like state movement and duplicate detection for aggregation operators. In addition, such a migration might take up to to w time intervals before the old instance can be stopped, where w represents the size of the migration window.

Storm [27] provides an abstraction of *task*, *executor*, *worker* and *topology*. This is illustrated in Figure 3, which shows a topology consisting of three workers. Each worker runs multiple executors, which host one or more tasks. The user defines the desired level of parallelism by defining the number of tasks for an operator. The system then tries to balance the tasks among all workers in the topology, based on the number of tasks assigned to each worker. Storm has also an option to rebalance the load during runtime to an increased or decreased number of hosts. This is realized by pausing and restarting the complete topology without loosing the operator state.

Similarly, Apache S4 [30] supports the notion of multiple processing nodes within a cluster, which allows a distributed execution. It can be customized using different load balancing or load shedding techniques, however the number of processing nodes in this system is fixed.

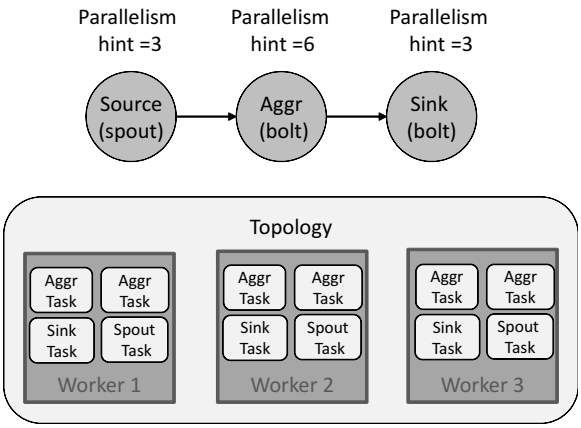

Figure 3: Distributed setup of Storm

In contrast, elastic data stream processing systems like StreamCloud [18] and SEEP [15] can dynamically vary the number of processing nodes within the system based on the workload. Therefore, both systems use a centralized architecture monitoring performance metrics like CPU utilization of each hosts (see Figure 4). The system requests new resource from the cloud resource manager as needed. Based on thresholds for individual hosts or the whole system, SEEP or StreamCloud decide to scale in or out respectively. The operator placement strategies are very simplistic, but adaptive. The only metric used for placement decisions is system load.

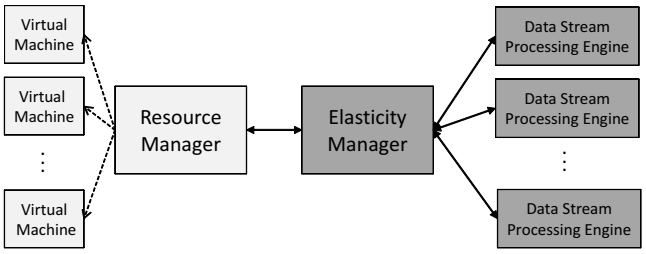

Figure 4: General architecture of an elastic data stream processing engine

StreamCloud [18] authors analyzed different design aspects for the design of an elastic scaling data stream processing engine. Based on their analysis, the best strategy for operator migration ensuring a minimal overhead towards the latency is based on a pause & resume approach. In contrast, SEEP integrates the mechanisms for load balancing and fault tolerance within a single prototype using enhanced state management mechanisms.

MillWheel [3] is able to partition the running operators using a dynamic key-based partitioning to a varying number of computational nodes. They partition the key space into a set of intervals, which can dynamically be split or merged. Similarly to SEEP and StreamCloud, decisions are based on a per-process measurement of the CPU load.

4. FAULT TOLERANCE MECHANISMS

Traditionally, distributed systems considered fault-tolerance as a fundamental requirement to be satisfied through the adoption of techniques, that were mostly designed to impose a lightweight load during best-case (i.e. no faults) executions. In such systems faults were considered as an exception that justified paying possibly severe performance degradation during a recovery phase. The advent of large-scale applications running on cloud platforms completely changed this scenario imposing fault-tolerance as first-class citizen in the design phase. In such systems, in fact, due to their sheer size, faults are no more exceptions but rather normal events that must be possibly dealt with during a vast majority of the executions. Data streaming systems aimed at cloud platforms are no exception, and must thus be designed to efficiently cope with faults. Faults may appear in a data streaming system at several different places. However, we can generally assume that a fault will either affect a single message (e.g. an overloaded network link discarding packets) or a computational element (e.g. an operator becomes unavailable after the crash of the CPU it was running in). As a consequence, we can differentiate between *State management* techniques employed to make operator state survive faults and *Event tracking* techniques employed to track the correct handling of events injected in the system.

4.1 State Management

State management for data stream processing systems is fundamentally based on the two classic approaches used to build fault tolerant software services: *active* and *passive replication*.

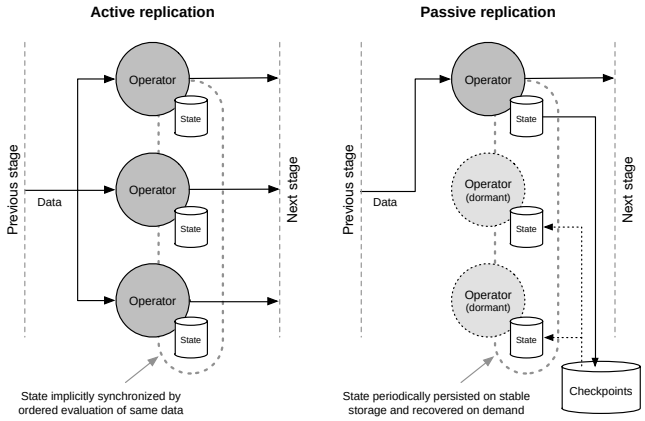

Figure 5: Basic functioning scheme for active (left) and passive (right) replication.

Active replication [6, 26] requires the system to evaluate an event in parallel on k identical operators (where k is the *replication degree*) such that the operator state is correctly maintained as long as less than k operators fails at the same time (see Figure 5 left). In order for this technique to work correctly (1) operator replicas must be guaranteed to evaluate the same set of events in the same order and (2) they must implement deterministic computations[2]. While active replication is considered very effective in providing uninterrupted service in the presence of faults, it is generally regarded as an inefficient technique for data stream processing systems. It requires for the execution of an application, in fact, k-times the resources that would be used by a corresponding non fault-tolerant execution. Furthermore, the deterministic fault-tolerance guarantees are more difficult to attain in a virtualized environment (like a cloud platform is) where the placement of operators on hardware machines is decoupled by the usage of multiple virtualization mechanisms.

Passive replication [20] is a *lazy* technique where operators are required to periodically backup their state to make it survive possible future faults (See Figure 5 right). Upon a fault, a new copy of the failed operator is instantiated and its state restored from the backup. The advantage of this approach is that few resources are periodically consumed at runtime to backup operator state, while most of the load is incurred only when an operator fails. In this case, in fact, beside restoring the latest operator internal state backup, all events managed by the failed operator before its failure that were not included in the backup must be evaluated again. Passive replication is today the most commonly used approach for providing operator fault tolerance in data stream processing systems, because it can easily be adapted and is tailored to work efficiently with different system architectures. In particular, most modern proposals adopt variants of this approach that can be mostly distinguished on the basis of the approach used to store state backups.

Storm does not provide state management explicitly. Its extension called Trident [28] includes interfaces for backing up operator states but leaves the actual implementation to the user.

Apache S4 [30] takes a lightweight approach to the problem as it assumes that lossy failovers are acceptable. Operators (called *Processing Elements* in S4) hosted on failed servers *Processing Nodes* are automatically moved to a standby server. Running operators periodically checkpoint their internal state through a mechanism that is both *uncoordinated* and *asynchronous*. It is uncoordinated because distinct operator instances can checkpoint their state autonomously without synchronization, and thus without global consistency. Moreover, checkpointing is executed asynchronously by first serializing the operator state and then saving it to stable storage through a pluggable adapter. When a new operator instance is created after a failure, first the system checks if a state for that operator has been previously saved to stable storage and, in this case, retrieves and restores it. S4's checkpointing mechanism can be overridden with a user implementation that could possibly provide stronger consistency guarantees.

Differently from S4, MillWheel [3] provides a simple programming model by guaranteeing strong consistency among all computations. This is achieved by checkpointing on persistent storage (based on BigTable [12]) every single state change incurred after a computation. Depending on the specific characteristics of the application that must be run, checkpointing can be either executed *before* emitting results downstream, or *after* the result emission. In the former case, operator implementations are automatically rendered idempotent with respect to the execution, greatly simplifying the programmer work, but potentially incurring greater latencies. In the latter case, it is up to the programmer making operators idempotent or ignoring the problem if the application allows it.

[2]Application-specific code may allow the correct execution and replication of non-deterministic computations [32].

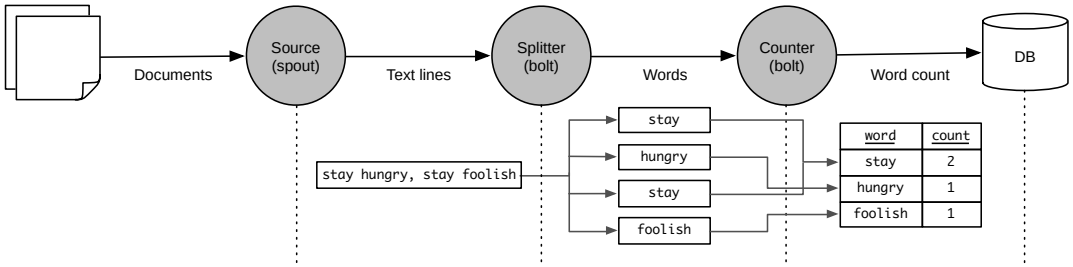

Figure 6: Example of event tree tracking in Storm

Timestream [33] also employs a shared reliable storage to backup information that are required for both fault recovery and dynamic reconfiguration of the DAG. For each operator, this information comprises two kinds of data dependencies: *state dependency*, that is the list of input events that made an operator reach a specific state, and *output dependency*, that is the list of input events that made an operator produce a specific output starting from a specific state. Such dependencies allow to correctly recover from a fault without having to store all the intermediate events produced by the operators and their states, since they can be recomputed by simply replaying required input events. Anyway, Timestream provides optimizations including the possibility to periodically checkpoint an operator state in order to avoid re-emitting the whole history of input events in order to recompute it.

While Apache S4, MillWheel and Timestream assume the presence of a storage backend where state can be persisted, SEEP [15] takes a different route allowing state to be stored on upstream operators. This choice, often considered inefficient, allows SEEP to treat operator recovery as a special case of a standard operator scale-out procedure, greatly simplifying system management. Operator state in SEEP is characterized by three information categories, namely *internal state*, *output buffers* and *routing state*, that are treated differently to reduce the state management impact on system performance.

The way D-Streams [42] manages computation state strictly depends on the model it uses for the computation itself. A D-Streams computation is structured as a sequence of deterministic batch computations on small time intervals, and the input and output of each batch, as well as the state of each computation, are stored as reliable distributed datasets (RDDs). For each RDD, the graph of operations used to compute (its *lineage*) it is tracked and reliably stored, so as to enable to recompute it in case of faults. As lineages are backed in a distributed fashion, the recovery of an RDD can be performed in parallel on separate nodes in order to speed up recovery operations. Furthermore, similarly to Timestream, in order to make recovery even faster, RDD state is periodically checkpointed to limit the number of operations required to restore it.

4.2 Event Tracking

When a fault occurs, events may need to be replayed. Event losses are detected by setting timeouts on the reception of acknowledgments from downstream operators, and the solution simply consists in making upstream operators replay lost events. When recovering from a failure, after the restoration of a previously checkpointed state, some events may need to be replayed to correctly restore the updated state. In order to understand which events have to be re-emitted, some information have to be tracked and managed at runtime.

Dealing with event losses boils down to decide (1) who is in charge of storing events that have not been acknowledged yet and (2) who is in charge of detecting losses and consequently triggering event replaying. In Storm [27], dedicated operators (*ackers*) are in charge of dealing with event losses by leveraging management messages sent by the other operators about the outcome of their event sendings. For each single input event, the ackers track the tree of events generated by the elaboration of such event in order to identify the input event to replay in case an operator notified a fault in a delivering. Figure 6 shows a simple example of the tracking information kept by an acker. The well known word count problem is used and the related topology is drawn on the top of the figure, with a Source node emitting one event for each line of the input documents, a Splitter node that emits an event for each word in each line, and finally a Counter node in charge of aggregating by word and providing the final output. On the bottom of the figure, the tree of the dependencies among the events generated by a single input event (a line of a document) is reported. Each event emitted by the Splitter (a word) is anchored to the input event, and in turn each event produced by the Counter (a word and the related count) is linked to an event emitted by the Splitter. Whenever the delivering of an event is not acknowledged within a predefined timeout, the ackers can easily understand which input event to replay (the root of the dependency tree). This solution requires input sources to keep the events they have produced until some acker tells them that such events are no longer required because their elaboration completed. Furthermore, it entails a relevant exchange of messages among operators and ackers to keep the tracking updated.

In the other systems that implement fault tolerance mechanisms (MillWheel [3], Timestream [33], SEEP [15] and D-Streams [42]), the event tracking functionality is strongly coupled with the state management because a checkpointed state has to be integrated with the replaying of the events emitted after such checkpoint in order to correctly recover the state itself. At this regard, events cannot be simply garbage collected when they are acknowledged, rather the checkpointing of a state updated with such events has to be waited for.

MillWheel [3], Timestream [33] and SEEP [15] do not employ any dedicated entity to manage event losses and simply designate upstream operators to track sent events until they get acknowledged, and to replay them when a timeout expires. In particular, Timestream does not store all the events and recompute those to be replayed by tracking their dependencies

with input events, similarly to Storm, and checkpointed states (state and output dependencies). In addition, Timestream supports an optimization that makes operators themselves temporarily buffer output events in order to avoid recomputing them from the scratch. While SEEP stores events and tracking information at the upstream operator, MillWheel and Timestream employ a reliable distributed store in order to tolerate faults where also an upstream operator of a failed downstream operator fails.

The event tracking in D-Streams [42] differs from the described approaches. D-Streams divides the computation in discrete intervals where events are moved from an interval to the next one by storing them in a reliable store composed by several reliable distributed datasets (RDDs). Each RDD is in charge of tracking its lineage, that is the graph of operations used to compute it.

5. FUTURE RESEARCH DIRECTIONS

Beside the recent great advancements in the field of data stream processing systems outlined in the previous sections, a few issues remain to be solved to fully adapt these systems to the peculiarities of cloud platforms. In particular, the following point describe research directions where we think further investigation would probably bring important improvements to the current state of the art.

Infrastructure awareness. Most existing data stream processing systems are infrastructure oblivious, i.e., their deployment strategies do not take into account the peculiar characteristics of the available hardware infrastructure. The physical connection and relationship among infrastructural element is known to be a key factor to both improve system performance and fault tolerance. For example the Hadoop system allows system administrators to manually define the association between computing elements and racks to make this information available to the scheduler (this feature is, in fact, called "rack awareness"). We think infrastructure awareness is an open research field for data stream processing systems that could possibly bring important improvements.

Cost-efficiency. Users in these days are not only interested in the performance of the system, but also the monetary cost of the used cloud-based infrastructure [16]. Therefore, cloud-based data stream processing systems must seek to minimize the monetary cost for the user. This includes an efficient scaling behavior maximizing the system utilization as well as efficient fault tolerance mechanisms. Bellavista et al. [8] recently present a first prototype, which allows the user to trade-off monetary cost and fault tolerance. Their prototype only selects a subset of the operators for replication based on a user-defined value for the expected information completeness. Furthermore, as big data-centers strive to go "green" improving their energy efficient, it is important to consider this as a further variable in the game.

Advanced elasticity. Although, several mechanisms for elastic scaling data streaming systems exists, the applied load balancing schemes are very simplistic. The used operator placement algorithms only optimize the utilization of the systems, other metrics like end to end latency or network bandwidth are only partially considered. However, production-scale systems often use these metrics to check contract-binding SLAs. Further research in this direction could possibly lead to systems where specific performance points can be probabilistically guaranteed.

Integration of different engines. Each of the presented prototypes is more or less suited for a specific use case. Given the increasing variety of different use cases, components automatically selecting the best engine would significantly improved the user experience. Early work [25, 14] on this topic showed promising results for such a federated approach. Duller et al. [14] presented an abstraction layer based on OSGI, which allows a federated execution on top of different streaming engines. Lim et al. [25] presented an optimizer component, which selects the best execution engine for a give query.

6. ACKNOWLEDGMENTS

This work has been partially supported by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research and by the academic project C26A133HZY funded by the University of Rome "La Sapienza".

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[2] 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. In *VLDB*, pages 120–139, 2003.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at internet scale. In *VLDB*, pages 1033–1044, 2013.

[4] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *DEBS*, pages 207–218, 2013.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, pages 50–58, 2010.

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

[7] R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *EDBT*, pages 919–930, 2006.

[8] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In *EDBT*, pages 85–96, 2014.

[9] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency

data processing with StreamMapReduce. In *CloudCom*, pages 48–58, 2011.

[10] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *ICDE*, pages 255–266, 2011.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, pages 668–668, 2003.

[12] 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 TOCS*, 2008.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, 2010.

[14] M. Duller, J. S. Rellermeyer, G. Alonso, and N. Tatbul. Virtualizing stream processing. In *Middleware*, pages 269–288, 2011.

[15] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.

[16] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *ACM Sigmod Record*, pages 43–48, 2009.

[17] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. Elastic scaling for data stream processing. *IEEE TPDS*, 2013.

[18] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, pages 2351–2365, 2012.

[19] M. Hirzel. Partition and compose: Parallel complex event processing. In *DEBS*, pages 191–200, 2012.

[20] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, pages 176–185, 2007.

[21] IBM. Dublin city council - traffic flow improved by big data analytics used to predict bus arrival and transit times. `http://www-03.ibm.com/software/businesscasestudies/en/us/corp?docid=RNAE-9C9PN5`, 2013.

[22] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD*, pages 431–442, 2006.

[23] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[24] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, pages 50–60, 2008.

[25] H. Lim and S. Babu. Execution and optimization of continuous queries with cyclops. In *SIGMOD*, pages 1069–1072, 2013.

[26] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. In *SRDS*, pages 21–30, 2011.

[27] N. Marz. Storm: Distributed and fault-tolerant realtime computation, 2012.

[28] N. Marz. Trident tutorial. `https://github.com/nathanmarz/storm/wiki/Trident-tutorial`, 2013.

[29] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, et al. The Niagara internet query system. *IEEE Data Eng. Bull.*, pages 27–33, 2001.

[30] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.

[31] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, pages 49–49, 2006.

[32] D. Powell. Delta4: A generic architecture for dependable distributed computing. *ESPRIT Research Reports*, 1, 1991.

[33] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Eurosys*, 2013.

[34] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[35] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS*, pages 1–12, 2009.

[36] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *PACT*, pages 53–64, 2012.

[37] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[38] Twitter. Improving twitter search with real-time human computation. `https://blog.twitter.com/2013/improving-twitter-search-with-real-time-human-computation`, 2013.

[39] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.

[40] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.

[41] Y. Yang, J. Kramer, D. Papadias, and B. Seeger. Hybmig: A hybrid approach to dynamic plan migration for continuous queries. *IEEE TKDE*, pages 398–411, 2007.

[42] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[43] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.