Wiley WIRE
*Wiley Interdisciplinary Reviews* 2012; **00**:1–14
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/??

# Elastic Stream Processing in the Cloud

Waldemar Hummer, Benjamin Satzger, Schahram Dustdar

[1] *Distributed Systems Group, Vienna University of Technology, Austria*

SUMMARY

Stream processing is a computing paradigm that has emerged from the necessity of handling high volumes of data in real time. In contrast to traditional databases, stream processing systems perform continuous queries and handle data on-the-fly. Today, a wide range of application areas relies on efficient pattern detection and queries over streams. The advent of Cloud computing fosters the development of elastic stream processing platforms which are able to dynamically adapt based on different cost-benefit tradeoffs. This article provides an overview of the historical evolution and the key concepts of stream processing, with special focus on adaptivity and Cloud-based elasticity. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Stream Processing, Event Processing, Cloud Computing, Elasticity

## 1. INTRODUCTION

Modern IT systems are handling an ever increasing volume of data, continuously generated by data producers such as social networks, electronic commerce systems, or Smart Cities, amongst others. These data streams include relevant information that can be revealed by data mining and processing. The traditional approach towards handling and analyzing data is to persist the data in a database and execute queries against it. However, there are scenarios in which not all the data can be stored because of high volume, or where real-time processing is required to react in a timely manner. Stream computing allows to process high loads of transient data in real time. Compared to traditional database management systems where queries are sent to the data, in stream computing the data are applied to continuously executed queries. Since operators typically do not have any control over the rate at which events are created, stream computing platforms need to be able to adapt. The number of events at low activity periods can be dramatically different from peak periods.

Over the last years, the cloud computing paradigm [1] has found widespread adoption. The reason for the success of cloud computing is the possibility to use services on-demand with a pay as you go pricing model, which proved to be convenient in many respects. One popular approach to cloud computing is the Infrastructure-as-a-Service [2] (IaaS) model, where virtual computing resources (virtual machines) are acquired and released on demand. This idea has been made popular in recent years by widely used implementations, such as Amazon's Elastic Compute Cloud (EC2) [3] or the OpenStack [4] open source Cloud middleware implementation. Cloud computing appears to be the perfect infrastructure for realizing an elastic stream computing service, by dynamically adjusting used resources to the current conditions. In economics, elasticity describes how the change in one variable (e.g., price) influences the change in another variable (e.g., demand). Elasticity

*Correspondence to: Distributed Systems Group, Vienna University of Technology, Argentinierstrasse 8/184-1, A-1040 Vienna, Austria in computing defines how computing resources are varied dynamically in order to cope with environmental changes, e.g., different workloads, variations in the price of computing resource, and changing user requirements regarding quality of service [5, 6, 7].

### *1.1. Methodology and Roadmap*

In this article we perform a systematic review of stream processing, with a focus on how elastic computing of data streams can be achieved on top of Cloud computing. The study attempts to put the topic into perspective under its evolutionary context, focusing on the one hand on the ground-breaking contributions which build the foundation of state-of-the-art stream processing, and discussing on the other hand the most recent developments that shape the future of the field.

We have thoroughly collected an initial set of over 100 research papers from scientific workshops, conferences, and journals, mostly published within the past 10 years. The papers were filtered based on the relevance and impact (number of citations) considering their age. Approximately half of the initially selected papers are included in this article. The authors also included some of their own related research work, but have carefully avoided to overemphasize it in the discussion.

The remainder of the paper is structured as follows. Firstly, Section 2 introduces the core concepts and terminology related to Cloud and stream processing. Section 3 discusses the emergence of the most important stream processing systems, providing a historical outline of the early days and rise of the research field. In Section 4, our focus shifts towards adaptivity and different strategies (mostly dating to the pre-Cloud era) for dealing with overload and other exceptional situations. Contrasting the approaches with largely static supply of resources in Section 4, Section 4.3 concentrates on resource-centric adaptation and discusses the relatively young research area of elastic stream processing in the Cloud. Section 5 summarizes the main findings and concludes the paper.

## 2. CORE PRINCIPLES OF STREAM PROCESSING

Based on existing models for event and stream computing (most notably [8, 9, 10]), we define the most important terms related to stream computing and event-based systems. The core artifacts and terminology are illustrated in Figure 1. Various research sub-areas use slightly different terminology [11], hence Figure 1 contains alternative terms (in brackets) for the core concepts. These terms are used interchangeably throughout the paper.

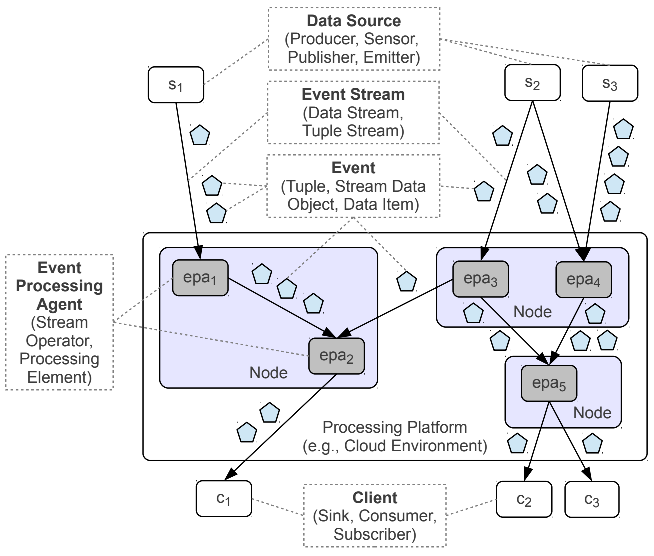
Data Source (Producer, Sensor, Publisher, Emitter)
Event Stream (Data Stream, Tuple Stream)
Event (Tuple, Stream Data Object, Data Item)
Event Processing Agent (Stream Operator, Processing Element)
s1
s2
s3
epa1
epa2
epa3
epa4
epa5
Node
Node
Node
Processing Platform (e.g., Cloud Environment)
c1
c2
c3
Client (Sink, Consumer, Subscriber)

Figure 1. Core Artifacts and Terminology

An *event* is an object encoding something that is happening for the purpose of computer processing (e.g., stock tick message). Typically, events are of a certain *type*, have a *timestamp*, and hold further, more specific data. A *complex event* results from applying processing steps, like aggregation and filtering, to one or more other events. Events are emitted by (event) *sources* and consumed by (event) *sinks*. An (event or data) *stream* is a linear sequence of events, usually ordered by time. Streams are usually considered (potentially) infinite, and a *window* [12, 13, 14] is some finite portion of a stream.

To distinguish different types of windows (or window queries), we formalize an event stream $E$ as a sequence of events $E = \langle e_1, e_2, e_3, \ldots \rangle$. A window query, at each point in time, evaluates a set of active (or open) windows, denoted $W$. A window $w \in W$ is a subsequence of the event stream, denoted $w \subseteq E$ (based on the notation for subsets). Each window $w$ has a start condition ($s(w)$) and an end condition ($e(w)$). The window types are illustrated in Figure 2 based on a simple exemplary event stream with "start" and "end" event types. One basic type of query window is the *growing window* which binds data values to be available over the entire stream. Another simple type is the *single item window* where each single event is considered separately. The single item window is particularly suitable for *stateless* operators, which only consider one event at a time and do not need to store the state of previous items. In *tumbling window* queries, new windows are only created if there is currently no other open window (i.e., $|W| = 0$). With a *sliding window* query, a new window is always opened if $s(w)$ holds. Finally, in a *landmark window* query, once a window has been opened, it remains open indefinitely (until the end of the stream is reached or the query is explicitly closed). In contrast to the growing window which binds some values over the entire stream, landmark windows consider different portions of the stream over time. More specialized characterizations of window types (e.g., time-based or count-based) are discussed in [13].

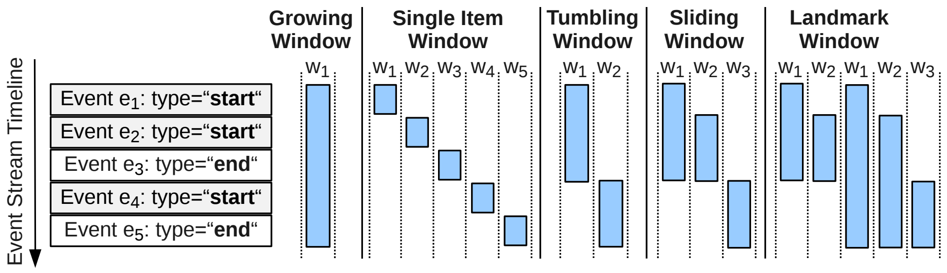

Figure 2. Different Types of Event Stream Query Windows (based on [14])

*Event processing agents* (EPAs) are software modules that consume events, process them, and output new events. Their behavior is defined by an *event processing language*, i.e., a high-level language, for instance based on an SQL dialect. *Stream processing*, *stream computing*, and *complex event processing* (CEP) are synonyms for performing computations with event streams as input. The behavior of the processing is define by an *event processing network* (EPN), a directed graph consisting of EPAs (as vertices) and event channels (as edges); the latter define the flow of events. While an EPN solely defines the logical connection between EPAs, the physical deployment is achieved by mapping EPAs to concrete computing *nodes*. A *data stream management system* (DSMS) is a software system whose main responsibility is the execution of one or multiple EPNs. It has responsibilities comparable to a traditional database management system (DBMS), with some notable differences, as shown in Table I.

The main challenges for stream processing are founded in the variability, unpredictability, burstiness, and volume of stream rates and data characteristics. Further aspects like changing processing objectives, e.g., higher quality of data requirements, and changing environmental circumstances, e.g., increased cost of computing resources, reinforce the challenge. Cloud computing and IaaS provide a new level of flexibility in resource management, which provides a solid basis for implementing highly elastic stream computing.

| DBMS | DSMS |
|---|---|
| Persistent relations | Transient streams |
| One-time queries | Continuous queries |
| Random access | Sequential access |
| Optimized access plans can be derived | Unpredictable data characteristics and arrival patterns |

Table I. Comparison of DBMS and DSMS

## 3. IMPORTANT SYSTEMS

This section discusses some of the most seminal contributions to the field of stream processing, which started to take off around the turn of the millennium. The ground-breaking early works are summarized in Section 3.1, before we shift the focus to more recent, largely distributed or Cloud-based approaches in Section 3.2.

### 3.1. *The Rise of a New Research Field*

The *NiagaraCQ* [19] continuous query system is among the first and most influential stream processing platforms. The NiagaraCQ command language, which allows to add and remove continuous queries at runtime, is tailored to XML data and based on the existing XML query language XML-QL[†] (whose features were later incorporated into the XQuery language [‡]). The basic form of a continuous query in NiagaraCQ is printed in Listing 1. One advantage of the NiagaraCQ query approach is that queries written in traditional languages can be easily transformed into continuous queries. NiagaraCQ also introduced the concept of incremental group optimization, which allows queries that are grouped together for performance reasons to be incrementally updated.

```
CREATE CQ_name
XML-QL query
DO action
{START start_time} {EVERY time_interval} {EXPIRE expiration_time}
```

Listing 1: Continuous Query in NiagaraCQ

At roughly the same time, Stanford introduce their seminal *STREAM* platform and discuss models and issues in data stream systems [12], such as (potentially) unbounded memory requirements, approximate query answering, window queries, and blocking operators such as aggregate functions. In contrast to XML-based NiagaraCQ, STREAM utilizes an extension of SQL as query language.

Also the *TelegraphCQ* [20] dataflow system was developed with the motivation that database systems at the time provided insufficient support for continuous adaptive querying, shared processing and different sources of unpredictability. TelegraphCQ is built on top of the open source DBMS PostgreSQL and the query syntax is based on SQL.

The aforementioned systems are largely based on declarative query languages which hide the query plan and internal processing logic from the user. The *Aurora* project [21, 22, 23] introduces a novel querying approach based on an explicit network graph with operator nodes and data flow connections (i.e., explicit modeling of the EPN).

### 3.2. *Road Towards Distributed Stream Processing*

To account for an ever increasing volume of data, research has put an emphasis on distributed and scalable stream processing. The *Medusa* [24] infrastructure, used as an extension to Aurora, integrates autonomous participants to process streaming data collaboratively. A currency-based market mechanism and other economic principles are applied to regulate participant collaborations. The distributed collaboration model introduces challenges such as naming, discovery, routing, or

[†]http://www.w3.org/TR/NOTE-xml-ql/
[‡]http://www.w3.org/TR/xquery/

message transport. Load management is achieved by repartitioning Aurora networks, typically by splitting (and remapping) operator nodes. The underlying principle that stream operators can be divided into atomic units is similar to the famous map-reduce paradigm, which later gained wide-spread popularity for batch processing of large data sets [25].

The *Borealis* [26] engine inherits the core functionality from Aurora and Medusa, and provides advanced capabilities for dynamism. Besides dynamic modification of running queries, Borealis allows revision of previously issued query results to recover from mistakes or problems with the input data. Revisions may originate from different sources: 1) from the input data (e.g., a stock ticker event corrects a previously published quote), 2) from the processing platform (e.g., events were dropped or deferred due to high load), or 3) from "time travel". Traveling back and forth in time over the input data sequence is a central concept in Borealis. Backwards travel means to rewind the input by issuing revision messages that allow the stream operators to reconstruct a certain historical state. Forward travel requires to predict the ongoing progression of the data streams, which evidently can lead to incorrectness and the necessity to revise results. Borealis also focuses on distributed processing via local, neighborhood and global optimization strategies. Since scheduling and placement of operators does not scale on a per-message basis, the notion of train scheduling [27] is introduced.

*System S* and its various satellite and successor projects (SPC [28], SODA [29], SPADE [30], InfoSphere Streams [31]) is an ongoing industry-driven research effort by IBM. SPC implements the general platform, provides a global event type system, and defines the notion of *processing elements*. SODA introduces an "importance" measure, which is typically based on a quantity or quality of the output stream, to maximize the utility of the data flow graphs. The SODA approach targets systems that are frequently rather than rarely overloaded, and since load shedding is not sufficient in such systems, SODA's scheduler entirely rejects low-priority jobs. SODA divides the global scheduling optimization into macro phases (determine admitted jobs and candidate nodes) and micro phases (fractional placement of operators on nodes). SPADE contributes an intermediate language and toolkit for stream operators and data flow graphs. The commercial InfoSphere platforms and its stream computing platform provide easy integration with the Cloud.

Microsoft's *Dryad* platform [32] is a general purpose distributed execution engine tailored to data-parallel applications. The Dryad runtime facilitates the implementation of inherently parallel applications by abstracting from standard concurrency mechanisms such as threads, scheduling, or synchronization. Although not a stream processing system in the narrower sense, Dryad has also influenced the field of online parallel data processing (e.g., [33]). The processing logic is modeled as a directed acyclic graph (DAG), and each vertex represents a sub-program which is mapped onto a physical resource. The DAG is specified based on a domain-specific language with notations for 1) creating new vertices (singleton graph; cloned sub-graph), 2) adding edges (pointwise composition; complete bipartite sub-graph), 3) merging graphs (concatenation of vertices; union of edges), and 4) customization and optimization (channel I/O types; encapsulation of vertices for scheduling; container vertices for output concatenation). Different graph refinements are implemented to optimize data locality and network usage.

The popular *Esper*[§] platform [34] provides a declarative language (Event Processing Language, EPL) and runtime library for event processing. EPL defines a comprehensive set of expressive language constructs for event correlation, filtering, and aggregation. The basic syntax of EPL has similarities with SQL (e.g., `select` statements), and the extended concepts range from window queries, pattern detection queries, or context-based queries to complex event hierarchies, splitting and duplicating of streams, predefined statistics views, and more.

Yahoo maintains the stream processing platform *S4* [35]. Processing elements in S4 are programmed with a fairly simply API (Application Programming Interface) consisting of methods `processEvent(..)` to consume data and `output()` to publish internal states to external systems.

The *Storm* platform [36] promises "extremely robust" support for distributed and fault-tolerant realtime computing, tailored to high-volume data streams. Storm has been in use for analytics

[§]http://esper.codehaus.org/

purposes by major companies like Twitter¶ or Groupon‖. Storm provides native clustering support and integrates seamlessly with Cloud providers like Amazon EC2. The storm daemon processes are designed to be stateless and fail-fast, hence daemons can be killed without affecting the overall health of the cluster.

## 3.3. *Comparison of Stream Processing Systems*

| **Platform** | **Year** | **Operation Definition** | **Main Focus** | **Core Achievements** |
|---|---|---|---|---|
| NiagaraCQ [19] | 2000 | Query based on XML-QL | Turn Passive Web into Active Environment | Timer-Triggered (Pull-Based) Continuous Web Queries; Grouping of Queries; Incremental Group Optimization |
| STREAM [12] | 2002 | Query based on SQL | From Persistent Relations to Transient Data Streams | Approximate Query Answering; Sliding Windows; Timestamping; Discussion of Algorithmic and Querying Issues |
| TelegraphCQ [20] | 2003 | Query based on SQL | Dealing with High-Volume, Highly-Variable Data Streams | Data Ingress and Caching; Adaptive Routing; Fjords (Push and Pull Inter-Module Communication) |
| Aurora [22] | 2003 | Operator Graph | Fundamental Architecture of DBMS for Streaming Data | Combining/Reordering Operator Boxes; Train Scheduling; Connection Points (Storing Stream Data); Ad-Hoc Queries; Stream Query Algebra; Load Shedding |
| Medusa [24] | 2003 | Operator Graph | Scalable, Distributed and Loosely Coupled Stream Processing | *Agoric* System; Autonomous Participants; Market Mechanism; Split & Remap; QoS Guarantees; Data Backup Control via Back Channels |
| Borealis [26] | 2005 | Operator Graph | Extend Aurora and Medusa with Critical Advanced Capabilities | Dynamic Query Modification; Time Travel; Result Revision; Flexible and Highly Scalable Monitoring and Optimization; |
| System S [28, 29, 30] | 2006 | Data-Flow Graph, Programming Language | Highly Scalable and Usability-Oriented Declarative Stream Processing | Data Fabric (Routing & Transport); Selective Job Admission and Scheduling; Stream Importance Weights; User-Oriented Abstraction Levels; Code Generation; Compiler Optimizations; Balanced Resource Allocation; Alternative Hardware Architectures (e.g., GPUs) |
| Dryad [32] | 2007 | Data-Flow Graph Connecting Sub-Programs | Generic Platform for Data-Oriented Parallel Programming | Automatic Scheduling and Distribution; DSL and Semantics for Construction of Data-Flow Graphs; Dynamic Graph Refinements; Efficient Job Pipelining; Processing Terabytes of Data in Minutes; Multi-Level Fault Tolerance |
| Esper [34] | 2007 | Event Processing Language (EPL) | Open-Source Event Processing Library for Java and .NET | Highly Expressive Query Language; Dynamic Event Type Definition; Pattern Matching Facilities; Actively Maintained, Production-Ready Library. |
| S4 [35] | 2010 | Programming API | Scalable, Partially Fault-Tolerant Platform with Simple Programming API | Keyed Data Events; Simple Processing Element API; Pluggable Architecture; Performance Evaluation of Lossy Failover |
| Storm [36] | 2012 | Programming API | Scalable, Fault-Tolerant Distributed Computation System | Guaranteed Processing; Automatic Reconfiguration with Stateless and Fail-Fast Daemons; Integration of Arbitrary Programming Languages; |

Table II. Comparison of Stream Processing Systems

The core differences of the stream processing systems discussed earlier in this section are summarized in Table II. For each platform, we record the year of the first seminal publication,

¶http://twitter.com
‖http://groupon.com

the type of notation employed to define the platform's operation, the main focus of the project as a whole, and a highlight of the keywords that express the core contributions and novelties achieved.

## 4. ELASTICITY AND ADAPTIVITY IN STREAM PROCESSING

One of the core challenges in stream processing is to adapt to highly dynamic environments, resource limitations, as well as variations in the data arrival (e.g., load bursts, out-of-order events) and data quality (e.g., uncertain/approximate data, result revisions). The work in [37] discusses pointedly a set of eight requirements (or guidelines) that should be tackled in real-time stream processing. Among these eight rules, three points are particularly interesting for elastic stream processing in the Cloud: handling of stream imperfections, guaranteed data safety and delivery, and automatic partitioning and scaling of applications. The proposed solutions in the literature can be roughly divided into strategies based on single events (Section 4.1) and more coarse-grained reorganization of the processing logic (Section 4.2). Some of the strategies discussed in this section are fundamental contributions that date back to the pre-Cloud era; more recent works have a stronger focus on Cloud and resource elasticity, which is discussed in Section 4.3.

### *4.1. Processing Strategies Based on Single Events*

The reliance on and uncontrollability of external data makes stream processing platforms inherently prone to overload situations or data inconsistencies like out-of-order event streams, resulting in the temporary or permanent inability of processing all incoming data. The major strategies for handling such exceptional conditions are discussed in the following. Note that the strategies are orthogonal and in fact often used in combination.

**Reordering** [38, 39] and **prioritization** [40, 41] are strategies which use an ordering function to give precedence to important results, and to reject or defer less important data or queries. Prioritization can be specified along with the query, or provided separately. In [40], query language extensions are introduced: `PRIORITY` and `DELIVERY ORDER` specify the local transmission order of results, and `SUMMARIZE AS` computes a *summary* of the query results, used to order the entire result set. In contrast to `ORDER BY` in standard SQL, with `DELIVERY ORDER` the score of each result can depend on all other tuples in the result. To keep the overhead of reordering at a minimum, best effort approaches that run concurrently with the processing are employed [38]. The *prefetch and spool* (P&S) technique fetches data from the input and sets aside uninteresting items to an auxiliary side-disk, denoted *spooling*. If an item on the side-disk becomes interesting (e.g, because ordering functions have changed), the input buffer is *enriched* with this item for further processing. *Index stride* [38] is used in online aggregation to partition the input streams (e.g., according to `GROUP BY`) and selectively fill the buffer with items from the currently underrepresented partition. Mechanisms for matching sequence patterns over out-of-order event streams are proposed in [42]. Depending on the likelihood of out-of-order event arrival, either an aggressive approach for maximum output or a conservative approach for guaranteed correctness is applied.

**Load shedding** [43, 44, 45, 22] is a well-studied mechanism for reducing the system load by dropping certain events from the stream. The simplest form is random drop, whereas semantic drop discards tuples with the lowest utility [22]. Three challenges can be identified [43]: when, where, and how much to shed load. These fundamental decisions can be expressed as an optimization problem, based on the data arrival rates and the capacity of the system, trying to reduce the input load while minimizing the loss of accuracy. The first challenge, detecting overload, is solved via a *load coefficient* which is computed based on the computational costs of the stream operators and their *selectivity*, i.e., the ratio of output rate to input rate. The latter two decisions are made based on a *load shedding roadmap* which determines the *(CPU-)cycle savings coefficient* at different points in the network. The amount of shedding performed before data arrive at an operator is denoted *sampling rate* [44]. In practice, load schedulers cannot take large amounts of time to dynamically compute optimal plans, hence offline algorithms are used to build a set of potential plans in advance [45].

In **deferred processing** [26, 22, 46], data that cannot be immediately handled are stored for later processing. The assumption of deferred processing is that the overload is limited in time, either for external reasons (e.g., the load burst is only temporary) or for internal reasons (e.g., the platform is about to acquire additional resources). In its simplest form, all types of incoming events are equally deferred, and more advanced implementations use prioritization where some outputs are given precedence over others. The utility of an output is determined by how much QoS is sacrificed if the processing is deferred [22].

### *4.2. Adaptation of Processing Logic*

Besides adaptation strategies for single events, more large scale reorganization of the processing logic and platform topology can be applied to cope with dynamic environments. The approaches discussed here have in common that the logic of the adaptation takes place on a more or less static set of computing resources, rather than considering elastic scale-out, which is the aim of Section 4.3.

The most obvious form of elasticity is to scale with the input data rate and the complexity of operations - acquiring new resources when needed, and releasing resources when possible. However, besides the mere computing power, more sophisticated issues must be taken into account. Most operators in stream computing are stateful and cannot be easily split up or migrated (e.g., window queries need to store the past sequence of events).

The performance of stream processing systems depends on the topology of operators, hence **efficient operator placement** [47] plays a key role. In [47], eight different operator placement algorithms are evaluated with respect to five core dimensions: node location (clustered/distributed), data rates (bursty/uniform), administrative domain (single/multiple), topology changes (dynamic/uniform), and queries (redundant/heterogeneous). The algorithm in [48] models the system load as a time series *X* and computes the load correlation coefficient $\rho_{ij}$ for pairs of nodes $i$ and $j$. The optimization goal is to maximize the overall correlation, which has the effect that the load variance of the system is minimized. A comprehensive and fine-grained model of CPU capacity and system load is provided in [49]. The *feasible set* of stream data rates under a certain placement plan is constructed. Mathematically, the feasible set corresponds to the (nonnegative) space under $n$ *node hyperplanes*, where $n$ is the number of nodes and the $i$-th hyperplane consists of all points that render node $i$ fully loaded. Another important goal is stabilization of the *buffer occupancy levels*, because EPAs can often take advantage of batch-processing several successive events [50] (to transfer data in larger units, decrease context-switching overhead, avoid memory cache misses, etc.). Within the highly scalable data streaming platform *StreamCloud* [51], query parallelization and operator placement play a central role. Based on an abstract query definition, StreamCloud materializes queries into complex networks of parallel operators, supporting intra-query parallelism as well as intra-operator parallelism.

An important design decision is whether the **adaptation logic** is **centralized or distributed**. In practice, construction of operator placement and data flow graphs cannot always be centralized, for instance because of size constraints or because the system is managed by multiple administrative domains. A distributed algorithm for deploying the dataflow graph in the network is presented in [52]. Nodes are hierarchically clustered (possibly in multiple levels) and one node per cluster is elected coordinator. The clustering allows a divide-and-conquer approach: the placement problem is split up into partitions for which locally optimal mappings are constructed. Given a dataflow graph $G$, the algorithm first exhaustively maps all vertices of $G$ to nodes within the top-level cluster. Each of the cluster nodes now contains a sub-graph of $G$, and the algorithm recursively refines the placement for each cluster node. In [53], the notion of *elastic operators* is proposed, where each operator has local control over its resource utilization. The elastic operator is based on an adaptive monitoring algorithm for assessing changes in the incoming workload. An alarm thread signals if the workload is peaking, which causes activation of additional resources (e.g., worker threads).

Most approaches discussed so far in this paper do not take into account the costs for **migrating to a new configuration**. However, relocation of EPAs at runtime is a non-trivial technical challenge and also resource intensive, particular if part of the state needs to be transferred. Network-aware dynamic operator placement, particularly for wide area networks, is discussed in [54]. The *minimum migration threshold* determines the required savings in network usage to amortize the reconfiguration costs for a specific placement. The *Flux* platform [55] was among the first to address the technical challenges of moving (migrating) stateful event stream operators. The proposed state movement protocol involves quiescing the input to the operator, transferring the state, and restarting the input stream in the new location. State migration also influences the load-balancing policy in Flux, which aims at achieving a unified utilization among the processing nodes while minimizing the number of state moves. In [56], a multi-query optimization algorithm is discussed which achieves a tradeoff between load distribution, duplicate event buffering and inter-node data traffic, also taking into account the costs of migration.

An important issue for reliable processing is **adaptation to cope with faults**. Handling faults such as crash of machines and loss of events has been an intensively studied challenge in stream processing. A comprehensive taxonomy of potential origins and effects of faults in event-based systems can be found in [11]. Among the proposed solutions are replica based approaches (redundant processing), upstream backup (restoring lost state from predecessor nodes), or checkpointing techniques (periodically forwarding the state from a primary node to a secondary fallback node) [15]. The challenge tackled in [57] is to minimize the loss of application output in the face of node failures. Each external input node is assigned an *importance value*, and the value of data at internal nodes (EPAs) is expressed as an aggregation of the importance of its inputs plus the "added value" generated by the preceding EPAs. The total expected output value includes the value of each node as well as the failure probabilities. The seminal work on fault tolerance in the Borealis system [58] introduces *DPC* (Delay, Process, and Correct) – a protocol for handling different types of failures in distributed stream processing. DPC allows the simultaneous crash of at most $n - 1$ of the $n$ replicas of each node. Fault handling mechanisms are also implemented in Dryad [32], where the data operators are transparently replicated by the platform and re-started in case of failure. To support fault tolerance, Dryad persists intermediate results and takes advantage of the fact that the processing graph is acyclic. For further details on fault models and fault handling, we refer the interested reader to related work in [11, 16, 15, 17, 18].

Finally, platforms are able to perform **adaptation of QoS** (or quality of data, QoD) by taking advantage of the fact that many applications do not require exact precision for stream-based queries. In [59], adaptive filters are utilized to balance the system load and quality levels. External inputs are subject to periodic shrinking by filters, and a precision manager is responsible to send out "growth messages" which ensure the demanded precision constraints.

## *4.3. Current and Future Trends for Elastic Stream Processing in the Cloud*

The Cloud computing paradigm introduces the possibility to easily allocate dynamic computing resources, hence fostering a highly elastic mode of operation. Elastic stream processing in the Cloud is a relatively young and arguably immature research area, which is evidenced by the fact that at the time of writing (early 2013) the scientific literature is mostly published in recent conference or workshop publications. In contrast to Section 4, where we discussed adaptation strategies based on a (more or less) static pool of resources, this section focuses on Cloud-based elasticity in stream computing. We identified four core challenges, which are discussed in the following paragraphs.

The advanced resource allocation possibilities provided by Clouds require **optimized integration** of stream engines with specific Cloud environments, and **novel (adaptation) algorithms** to integrate and exploit this new potential. We can anticipate that there is still potential for leveraging specific hardware characteristics, virtualization on different abstraction levels, and selective use of available cloud services. One of the most basic forms of elasticity in the Cloud is to use a pool of fixed resources for average loads and additional machines for peak loads, as discussed in [60]. *Nephele* [33] is among the first data processing platforms to exploit dynamic resource provisioning offered by IaaS clouds. The core deployment components are the job manager (job scheduling, resource management), a multitude of task managers (deployed on each node, manage task execution) and a Cloud controller (interface to the IaaS Cloud). A fine-grained CPU utilization analysis allows to employ specific virtual machine types for processing different tasks. *ESC* is an elastic stream processing platform for the Cloud [61] which scales seamlessly with the number of provided computing resources. ESC exploits lightweight processes provided by the Erlang[**] platform for transparent distribution of data processing jobs. Since elasticity can have a critical influence on the application functionality, we also regard the field of reliability as a key challenge for Cloud operators – regarding both fault resistance [62] and upfront testing of event-based data processing applications [63].

Following the recent trend towards *everything as a service* [64] (*XaaS*), **stream processing as a service** (or data processing as a service, *DaaS*) is certainly a grand challenge for the near future: How can Cloud service providers offer stream processing as a service to their customers? Which languages and tools are most appropriate to support different levels of user experience, and how are the service level objectives (SLOs) best defined and enforced on a per-tenant basis? Multi-tenancy also introduces pressing legal and practical challenges such as data security and privacy [65]. The *WS-Aggregation* [66, 56] platform takes a service-oriented approach to event processing with loosely coupled aggregator nodes that process streams collaboratively. The platform is tailored to multi-tenancy and optimizes resource usage through multi-query optimization. Streaming as a service is also targeted by the *Stormy* platform [67]. Stormy aims at providing elastic stream processing that scales with the number of queries as well as the number of streams. Another XaaS approach by [68] has been termed *continuous analytics as a service* (CaaaS). One of the main challenges in CaaaS is integration of stream data with persistent (stored) data, which is solved via an integrated query language that combines both paradigms. *Data staging* [68] is an important concept in CaaaS and data warehousing, allowing for stepwise archival of historical data. Because often only the latest data are important, young data objects can be put to high-performance in-memory caches, whereas older data can be stored to different types of persistent databases. The Cloud is utilized to obtain storage resources, with appropriate level of availability, access performance, and clustering.

The legal and business aspects of Cloud computing require many platform providers to integrate support for customized **service-level agreements** (SLAs), or **consistency guarantees**, combined with suitable **monitoring infrastructures** to ensure that SLAs are not violated. The famous *CAP theorem* [69] proposed by Eric Brewer states that any distributed system can satisfy at most two out of the three properties consistency, availability, and partition tolerance. This conjecture, which has also been formally proven valid [70], has fundamental implications on elastic stream processing in the Cloud: since network partitions can never be entirely ruled out (i.e., partition tolerance needs to be ensured), essentially developers are left with the choice between availability and consistency. In this context, the term *eventual consistency* [71] has coined database research in the previous years, also applied to data streams (e.g., in [72]). One remaining challenge is whether and how these two latter goals could potentially be combined for stream processing, e.g., by dynamically activating one or the other goal at runtime. For instance, the Stormy platform [67] claims to focus on both, availability and strong consistency. Events are processed with a configurable replication factor, and within each replication group events are guaranteed not to get lost and to be processed in total order.

Moreover, **explicit pricing information** becomes available with Cloud computing. One of the challenges with critical business impact is how to define the price-quality tradeoff, and how to support and enforce quality guarantees at runtime. Early works on SLAs for stream processing were mostly quality of data oriented, but the Cloud promotes a primarily cost-based view on QoS. Cost-based optimization of service compositions [73] is a related field and partly applicable as well, but stream processing entails more specific issues that need to be addressed. The generic cost model for wide-area stream processing discussed in [74] distinguishes operator costs, query costs, node costs and system costs (from finer to more coarse grained). Since the model is generic, it can be directly mapped to monetary costs associated with CPU utilization on a Cloud host. It should be noted that not only the current and desired configuration has to be considered for cost calculation, but also the steps required to reach the new state. Hence, the topology optimization in [56], which allows to migrate event streams between nodes, also takes into account the cost of migration. The work in [75] aims at minimizing the economic cost of using the Cloud, roughly distinguishing between

[**]http://www.erlang.org

*data parallel* and *task parallel* applications, which have different requirements and cost factors associated.

The fourth major open issue is how we can push the limits of stream computing to provide **efficient locality- and context-aware data delivery on a global scale**, integrating desktop machines, mobile devices, and federated data centers. Another aspect of this challenge is the increasing importance of green computing and energy-efficiency which has embraced and influenced various research areas. As today's Clouds become interoperable and more coordinated, an envisioned worldwide network of interconnected Cloud data centers needs to cope with new challenges [76]. Data locality is important to minimize latency, and computation should be as close to the source as possible. The *RACE* architecture proposed in [77] integrates a multitude of end devices organized in a star-like *Cloud-Edge* topology. Query graphs and operator placement plans are generated from queries expressed in LINQ[††] (Language Integrated Query). The *Mortar* platform [78] provides control over wide-area stream processing applications. Mortar uses multipath routing policies, while still guaranteeing duplicate-free processing. The authors advocate the use of a hierarchical overlay network with a primary tree and multiple sibling trees. The Mortar framework is also used in [79] to perform wide-scale ad-hoc data processing in the Cloud. The microblogging service Twitter uses *Storm* [36] to process tweet messages and user clicks for analytics purposes. The tuple stream based processing allows scalability in the order of one million 100 byte messages per second per node [36], collected from a worldwide user base. The Web Service Stream Deployer (WSSD) proposed in [80] is a component-based architecture for remote deployment of streams across widely distributed resources.

## 5. CONCLUSION

The ever increasing amount of data generated around the globe, paired with advancing requirements for continuous queries and online analytics, demand for highly efficient, scalable and reliable stream processing solutions. Stream processing has emerged as a research direction starting from the late 1990ies with a variety of seminal contributions achieved since then. Due to the uncontrollability of external data and the highly dynamic operating environment, state-of-the-art stream processing platforms are inherently geared towards adaptivity and elasticity. Early forms of adaptation include strategies based on single events or event streams (e.g., load shedding), as well as more coarse-grained reorganization of the processing platform (e.g., optimized operator placement). The advent of Cloud computing has fostered a resource-oriented view towards adaptation, allowing to elastically allocate and release resources on demand. Ongoing trends in technology, politics and society in general introduce high-priority issues such as privacy and security, energy efficiency, or data traceability. With some of these challenges yet seeking to be solved in the context of continuous data processing, elastic stream computing in the Cloud certainly remains an important and high-impact research field for the years to come.

## REFERENCES

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, *et al.*. A view of cloud computing. *Communications of the ACM* 2010; **53**(4):50–58.
2. Hilley D. Cloud computing: A taxonomy of platform and infrastructure-level offerings. *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-13* 2009; .
3. Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2` 2012.
4. Openstack. Openstack open source cloud computing software. `http://www.openstack.org/` 2012.
5. Voorsluys W, Broberg J, Buyya R. Cloud computing: Principles and paradigms 2011.
6. Dustdar S, Guo Y, Satzger B, Truong HL. Principles of elastic processes. *Internet Computing, IEEE* 2011; **15**(5):66–71, doi:10.1109/MIC.2011.121.
7. Dustdar S, Guo Y, Han R, Satzger B, Truong HL. Programming directives for elastic computing. *Internet Computing, IEEE* 2012; .

[††]http://msdn.microsoft.com/en-us/library/Aa479865.aspx

8. Moxey C, *et al.*. A conceptual model for event processing systems. *IBM Redguide publication* 2010; Http://www.redbooks.ibm.com/abstracts/redp4642.html.
9. Luckham D, Schulte R. Event processing glossary v1.1. *Event Processing Technical Society* 2008; **2**.
10. Sharon G, Etzion O. Event-processing network model and implementation. *IBM Systems Journal* 2008; **47**(2):321–334.
11. Hummer W, Inzinger C, Satzger B, Leitner P, Dustdar S. Deriving a unified fault taxonomy for event-based systems. *6th ACM International Conference on Distributed Event-Based Systems (DEBS'12)*, 2012; 167–178.
12. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. *21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, ACM, 2002; 1–16.
13. Patroumpas K, Sellis T. Window specification over data streams. *Current Trends in Database Technology - EDBT 2006*, 2006; 445–464.
14. Botan I, Kossmann D, Fischer PM, Kraska T, Florescu D, Tamosevicius R. Extending xquery with window functions. *33rd international Conference on Very Large Data Bases (VLDB)*, 2007; 75–86.
15. Kwon Y, Balazinska M, Greenberg A. Fault-tolerant stream processing using a distributed, replicated file system. *Proc VLDB Endow.* 2008; **1**(1):574–585.
16. Hwang JH, Balazinska M, Rasin A, Cetintemel U, Stonebraker M, Zdonik S. High-Availability Algorithms for Distributed Stream Processing. *21st International Conference on Data Engineering (ICDE)*, 2005.
17. Cristian F. Understanding fault-tolerant distributed systems. *Communications of the ACM* 1991; **34**(2):56–78.
18. Satzger B. Self-healing distributed systems. PhD Thesis, Ph. D. dissertation, Universität Augsburg, Germany 2008.
19. Chen J, DeWitt D, Tian F, Wang Y. Niagaracq: A scalable continuous query system for internet databases. *ACM SIGMOD Record*, vol. 29, ACM, 2000; 379–390.
20. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, *et al.*. Telegraphcq: Continuous dataflow processing for an uncertain world. *CIDR*, 2003.
21. Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S. Monitoring streams: a new class of data management applications. *28th International Conference on Very Large Data Bases (VLDB)*, 2002; 215–226.
22. Abadi D, Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 2003; **12**(2):120–139.
23. Balakrishnan H, Balazinska M, Carney D, Çetintemel U, Cherniack M, Convey C, Galvez E, Salz J, Stonebraker M, Tatbul N, *et al.*. Retrospective on aurora. *The VLDB Journal* 2004; **13**(4):370–383.
24. Cherniack M, Balakrishnan H, Balazinska M, Carney D, Cetintemel U, Xing Y, Zdonik S. Scalable distributed stream processing. *Proc. Conf. on Innovative Data Syst. Res*, 2003.
25. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
26. Abadi D, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang J, Lindner W, Maskey A, Rasin A, Ryvkina E, *et al.*. The Design of the Borealis Stream Processing Engine. *Conference on Innovative Data Systems Research (CIDR)*, 2005.
27. Carney D, Çetintemel U, Rasin A, Zdonik S, Cherniack M, Stonebraker M. Operator scheduling in a data stream manager. *29th International Conference on Very Large Data Bases (VLDB)*, 2003; 838–849.
28. Amini L, Andrade H, Bhagwan R, Eskesen F, King R, Selo P, Park Y, Venkatramani C. SPC: a distributed, scalable platform for data mining. *4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, ACM, 2006; 27–37.
29. Wolf J, Bansal N, Hildrum K, Parekh S, Rajan D, Wagle R, Wu KL, Fleischer L. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. *9th ACM/IFIP/USENIX International Conference on Middleware*, Springer, 2008; 306–325.
30. Gedik B, Andrade H, Wu K, Yu P, Doo M. Spade: the system s declarative stream processing engine. *ACM SIGMOD International Conference on Management of Data*, ACM, 2008; 1123–1134.
31. Biem A, Bouillet E, Feng H, Ranganathan A, Riabov A, Verscheure O, Koutsopoulos H, Moran C. IBM InfoSphere Streams for Scalable, Real-Time, Intelligent Transportation Services. *ACM SIGMOD International Conference on Management of Data*, ACM, 2010; 1093–1104.
32. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, EuroSys '07, 2007; 59–72.
33. Warneke D, Kao O. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**(6):985–997.
34. Bernhardt T, Vasseur A. Esper: Event stream processing and correlation. *ONJava, in http://www.onjava.com/lpt/a/6955, O'Reilly* 2007; .
35. Neumeyer L, Robbins B, Nair A, Kesari A. S4: Distributed stream computing platform. *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, IEEE, 2010; 170–177.
36. Project S. Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net/` 2012.
37. Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. *SIGMOD Record* 2005; **34**(4):42–47.
38. Raman V, Raman B, Hellerstein JM. Online dynamic reordering for interactive data processing. *25th International Conference on Very Large Data Bases (VLDB)*, VLDB '99, 1999; 709–720.
39. Avnur R, Hellerstein JM. Eddies: continuously adaptive query processing. *ACM SIGMOD International Conference on Management of Data*, 2000; 261–272.
40. Hull B, Bychkovsky V, Zhang Y, Chen K, Goraczko M, Miu A, Shih E, Balakrishnan H, Madden S. Cartel: a distributed mobile sensor computing system. *4th International Conference on Embedded Networked Sensor Systems*, ACM, 2006; 125–138.
41. Madden S, Franklin M. Fjording the stream: An architecture for queries over streaming sensor data. *18th International Conference on Data Engineering*, IEEE, 2002; 555–566.

42. Liu M, Li M, Golovnya D, Rundensteiner E, Claypool K. Sequence pattern query processing over out-of-order event streams. *25th International Conference on Data Engineering*, 2009; 784–795.
43. Tatbul N, Çetintemel U, Zdonik S, Cherniack M, Stonebraker M. Load shedding in a data stream manager. *29th International Conference on Very Large Data Bases*, VLDB Endowment, 2003; 309–320.
44. Babcock B, Datar M, Motwani R. Load shedding for aggregation queries over data streams. *20th International Conference on Data Engineering*, IEEE, 2004; 350–361.
45. Tatbul N, Çetintemel U, Zdonik S. Staying fit: Efficient load shedding techniques for distributed stream processing. *Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007; 159–170.
46. Zang C, Fan Y. Complex event processing in enterprise information systems based on rfid. *Enterprise Information Systems* 2007; **1**(1):3–23.
47. Lakshmanan G, Li Y, Strom R. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing* 2008; **12**(6):50–60.
48. Xing Y, Zdonik S, Hwang J. Dynamic load distribution in the borealis stream processor. *21st International Conference on Data Engineering (ICDE)*, IEEE, 2005; 791–802.
49. Xing Y, Hwang JH, Çetintemel U, Zdonik S. Providing resiliency to load variations in distributed stream processing. *32nd International Conference on Very Large Data Bases (VLDB)*, 2006; 775–786.
50. Amini L, Jain N, Sehgal A, Silber J, Verscheure O. Adaptive control of extreme-scale stream processing systems. *26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2006.
51. Gulisano V, Jimenez-Peris R, Patio-Martinez M, Soriente C, Valduriez P. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems* 2012; **23**(12):2351 –2365.
52. Kumar V, Cooper B, Schwan K. Distributed stream management using utility-driven self-adaptive middleware. *2nd International Conference on Autonomic Computing (ICAC)*, 2005; 3–14.
53. Schneider S, Andrade H, Gedik B, Biem A, Wu KL. Elastic scaling of data parallel operators in stream processing. *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2009; 1–12.
54. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M. Network-aware operator placement for stream-processing systems. *22nd International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 2006; 49–.
55. Shah MA, Hellerstein JM, Chandrasekaran S, Franklin MJ. Flux: An adaptive partitioning operator for continuous query systems. *19th International Conference on Data Engineering (ICDE)*, 2003; 25–36.
56. Hummer W, Leitner P, Satzger B, Dustdar S. Dynamic migration of processing elements for optimized query execution in event-based systems. *1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), OnTheMove Federated Conferences (OTM'11)*, 2011; 451–468.
57. Bansal N, Bhagwan R, Jain N, Park Y, Turaga D, Venkatramani C. Towards optimal resource allocation in partial-fault tolerant applications. *INFOCOM 2008. The 27th Conference on Computer Communications*, IEEE, 2008; 1319–1327.
58. Balazinska M, Balakrishnan H, Madden SR, Stonebraker M. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)* 2008; **33**(1):3:1–3:44.
59. Olston C, Jiang J, Widom J. Adaptive filters for continuous queries over distributed data streams. *SIGMOD International Conference on Management of Data*, ACM, 2003; 563–574.
60. Kleiminger W, Kalyvianaki E, Pietzuch P. Balancing load in stream processing with the cloud. *27th International Conference on Data Engineering, Workshops (ICDEW)*, IEEE, 2011; 16–21.
61. Satzger B, Hummer W, Leitner P, Dustdar S. Esc: Towards an elastic stream computing platform for the cloud. *2012 IEEE Fifth International Conference on Cloud Computing* 2011; **0**:348–355.
62. Zinn D, Hart Q, McPhillips T, Ludascher B, Simmhan Y, Giakkoupis M, Prasanna V. Towards reliable, performant workflows for streaming-applications on cloud platforms. *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011; 235–244.
63. Hummer W, Raz O, Shehory O, Leitner P, Dustdar S. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability* 2013; :(to appear).
64. Banerjee P, Friedrich R, Bash C, Goldsack P, Huberman B, Manley J, Patel C, Ranganathan P, Veitch A. Everything as a service: Powering the new information economy. *IEEE Computer* 2011; **44**(3):36–43.
65. Shi E, Chan THH, Rieffel EG, Chow R, Song D. Privacy-preserving aggregation of time-series data (ndss). *Network and Distributed System Security Symposium*, 2011.
66. Hummer W, Satzger B, Leitner P, Inzinger C, Dustdar S. Distributed continuous queries over web service event streams. *7th International Conference on Next Generation Web Services Practices (NWeSP'11)*, 2011; 176–181.
67. Loesing S, Hentschel M, Kraska T, Kossmann D. Stormy: an elastic and highly available streaming service in the cloud. *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, 2012; 55–60.
68. Chen Q, Hsu M, Zeller H. Experience in Continuous analytics as a Service (CaaaS). *14th International Conference on Extending Database Technology (EDBT)*, ACM, 2011; 509–514.
69. Brewer EA. Towards robust distributed systems. *19th ACM Symposium on Principles of Distributed Computing (PODC)*, vol. 19, 2000; 7–10.
70. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 2002; **33**(2):51–59.
71. Vogels W. Eventually consistent. *Communications of the ACM* 2009; **52**(1):40–44.
72. Sebepou Z, Magoutis K. Cec: Continuous eventual checkpointing for data stream processing operators. *41st IEEE/IFIP International Conference on Dependable Systems&Networks (DSN)*, 2011; 145–156.
73. Leitner P, Hummer W, Dustdar S. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing* 2011; **PP**(99):1.
74. Papaemmanouil O, Çetintemel U, Jannotti J. Supporting generic cost models for wide-area stream processing. *IEEE International Conference on Data Engineering (ICDE)*, 2009; 1084–1095.
75. Ishii A, Suzumura T. Elastic stream computing with clouds. *IEEE International Conference on Cloud Computing (CLOUD)*, 2011; 195–202.

76. Satzger B, Hummer W, Inzinger C, Leitner P, Dustdar S. Winds of change: From vendor lock-in to the meta cloud. *Internet Computing, IEEE* 2013; :(to appear).
77. Chandramouli B, Claessens J, Nath S, Santos I, Zhou W. RACE: real-time applications over cloud-edge. *ACM SIGMOD International Conference on Management of Data*, 2012; 625–628.
78. Logothetis D, Yocum K. Wide-scale data stream management. *USENIX 2008 Annual Technical Conference (ATC)*, USENIX Association: Berkeley, CA, USA, 2008; 405–418.
79. Logothetis D, Yocum K. Ad-hoc data processing in the cloud. *Proc. VLDB Endow.* 2008; **1**(2):1472–1475.
80. Martinaitis P, Patten C, Wendelborn A. Component-based stream processing in the cloud. *Workshop on Component-Based High Performance Computing*, ACM, 2009; 16.