Parallel and Distributed Stream Processing: Systems Classification and Specific Issues

Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, Yves Caniou

Parallel and Distributed Stream Processing: Systems Classification and Specific Issues

Roland Kotto-Kombi[1,2], Nicolas Lumineau[1,3], Philippe Lamarre[1,2], and Yves Caniou[1,4]

[1] Université de Lyon, CNRS
[2] INSA-Lyon, LIRIS, UMR5205, F-69621, France
[3] Université Lyon 1, LIRIS, UMR5205, F-69622, France
[4] ENS de Lyon, LIP, F-69364, France

Abstract. Deploying an infrastructure to execute queries on distributed data streams sources requires to identify a scalable and robust solution able to provide results which can be qualified. Last decade, different Data Stream Management Systems have been designed by exploiting new paradigm and technologies to improve performances of solutions facing specific features of data streams and their growing number. However, some tradeoffs are often achieved between performance of the processing, resources consumption and quality of results. This survey[5] suggests an overview of existing solutions among distributed and parallel systems classified according to criteria able to allow readers to efficiently identify relevant existing Distributed Stream Management Systems according to their needs ans resources.

Keywords: Distributed Stream Management Systems, Workflow, Map Reduce

1 Introduction

With the multiplication of data streams sources (sensor networks, connected devices...), stream analysis applications have known great evolution on last years. The treatment of data stream that is to request or to analyze data represents a challenge in terms of performance, scalability, robustness and results quality. Unlike disk-based data, data streams are potentially infinite and some of their features are unpredictable like items distribution and throughput. To request data streams, the solutions[21, 18] based on centralized DBMS proves to be limited and irrelevant[22, 2]. Thus, some *Data Stream Management Systems* (DSMS) were designed, like STREAM[4], Aurora[2] or TelegraphCQ[10], to compute queries on data streams, called *continuous queries*. These queries may be represented as dataflow diagrams where streams runs between operators to deliver results to end-users with low latency. Concerning the deployment of DSMS, they turn from centralized but multi-core systems running on

[5] This work has been partially supported by French National Agency for Research (ANR), project SOCIOPLUG (ANR-13-INFR-0003).

a single machine [2] to a distributed infrastructure like Grid or Cloud [1, 20]. It is worth noting that all systems we consider in this paper are able to execute multiple queries simultaneously and exploit on or more type of parallelism described in the course of this article.
With the apparition of MapReduce framework[12], a different way of distributing operators is developed. MapReduce has the advantage to provide a highly parallel programming paradigm and to remain simple. Operators are simple to define and parallelism management is hidden to users. In that context, some DSMS based on MapReduce framework have been developed like SparkStreaming[24], C-MR[8], M3[3], and provide different visions on stream processing.
This survey aims at exploring the various issues faced by existing Data Stream Management Systems, at bringing a more precise view of continuous query processing and at facilitating the comparison of the different known solutions.
In this article, we present an original and up-to-date classification of parallel DSMS. This classification is not only based on different paradigms on which DSMS have been defined these last years, but also considering some aspects related to systems capacities to optimize data computations by reusing intermediate results and to be deploy in a distributed way. These inherent DSMS features are completed by considering complementary aspects related to resource consumption, robustness and quality results.
The remainder of this paper is built as follow: in Section 2, the background exposes stream definitions and how they can be processed with workflows. The MapReduce framework is also reminded. Next, Section 3 presents the related work about existing surveys and their limits. Our classification of existing DSMS is proposed in Section 4. Section 5 proposes a transversal point of view of DSMS on aspects related to the resource consumption, the robustness and the quality of results, before concluding into Section 6.

2 Background

To understand the specific requirements of parallel and distributed stream processing, we remind readers about some basic notions like stream and window definitions. Next we briefly remind some features about operators and query language, before considering the two main paradigm relevant to improve stream processing performance: workflow and Map Reduce.

2.1 Stream

We remind the stream definition given in [5].

Definition 1. (Stream) *Let consider a schema S, composed by attributes which describe data, and an ordered timestamp set τ. A stream is a potentially infinite multiset of elements $< s_i , \tau_i >$ where s_i is a tuple of the stream respecting the schema S and $\tau_i \in \tau$ the associated timestamp.*

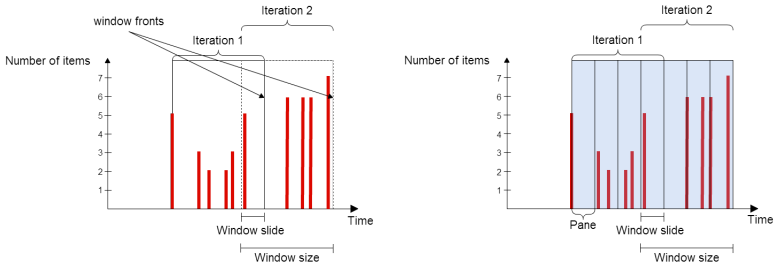
It is important to notice that the timestamp is not included in the schema so many stream elements can share the same timestamp and the number of elements sharing a same timestamp must be finite.
Data streams can not be managed and processed like static data because of specific features like unpredictability. For more details, see former surveys [7, 22]. Actually, they are potentially infinite bag of data. Data streams can not be stored on disk for further treatments because they may require unbounded amount of memory. From a practical point of view, data streams can not be gathered completely before processing them. Moreover, they are unpredictable considering input variations. Items may arrive by millions per seconds like click logs for website monitoring applications. Stream rate variations may happen any-time and require more resources during runtime. Finally, systems can not anticipate arrivals of data, as they arrive continuously, not at regular intervals.

2.2 Window

Because data streams can not be stored on memory, an alternative is to consider only recent data, called *computation window*, (or just window [7]) to get a result based on a data subset.

Definition 2. (Computation Window) *A computation window is a logic stream discretization [24] which is defined by a size and a slide (see Figure 1a). Considering the front of a window and the chronological order, the size defines the timestamp interval of elements to consider. The slide defines the step between two consecutive window fronts.*

According to Definition 2, a window is denoted *tumbling* window as found for instance in [13, 2] if size and slide values are the same. In other cases, it is denoted a *sliding* window as we may find in [8, 24].



(a) Sliding window

(b) Window division in multiple panes

Fig. 1: Computation window representation

For both types, the size and slide for window can be based on two different units of measurement: time-based and count-based approaches.

Definition 3. (Time-based window) *A time-based window is defined on a time interval t_0 to t_n. A stream element e belongs to the window if $\tau_e \in [t_0, t_n[$ with τ_e the timestamp of e.*

In Figure 1a, the first time interval, represented by iteration 1, allows to gather 20 first elements (5+3+2+2+3+5) together and iteration 2 allows to gather 30 elements (5+6+6+6+7) together. It is worth noting that considering the current window slide and window size, five elements are common for the both iterations.

Definition 4. (Count-based window) *A count-based window is defined according to a number of elements k. This value k corresponds to the window size. The k-th most recent elements belong to the current iteration. The slide defines the number of new elements to receive before computing a new result on the updated window content.*

As highlighted in [5], count-based windows are not deterministic when considering multiple elements per timestamp stream. Indeed, let consider a count-based window which considers the last one hundred stream elements. If more than one hundred elements arrive in the system at a same timestamp the window definition is not deterministic anymore because it is impossible to determine which elements to consider for computation. According to window definition (see Definition 2), we accept that consecutive iterations can share elements when the slide is smaller than the size. These shared subwindows, denoted *panes*, represent a logic stream discretization as illustrated on Figure 1b. Considering that window size and slide are fixed for all the computation, the size of a pane is mostly given by the greatest common divisor between the size and the slide of a given windowing schema. As mentioned above, a pane can be shared by many consecutive iterations. This means that the result of a query on a pane can be useful for all iterations it belongs. It is then a crucial feature to consider for *incremental* computation of sliding windows for mutualization of pane results.

2.3 Stateless or stateful operators

Operators (filter, join, sort ...) applied on data streams may require to treat element by element or by logical block of elements. Thus, we distinguish two main categories of operators : stateless and stateful operators.

Stateless operators, for example filters based on an attribute value, processed data streams element by element. They return a new result with an unpredictable frequency. For example, a filter will return nothing if its input does not satisfy the filtering predicate. Moreover, they do not have information about the current iteration of the computation window or former results when they compute a result from a data stream element. Nevertheless, a stateless operator may used historic data stored on local memory or disk. It allows to compute joins or some aggregate operators within a stateless operator.

In opposition, stateful operators takes as input a logic block of stream elements to compute a single result for the entire set. The stream element block may be defined by a range of timestamps or a number of elements to consider. They keep information like the current iteration of the window or the list of distinct element values to process a result from all considered elements. Information generated during a stateful operator runtime is denoted its *state*. For example, a window-based stateful operator computing the sum of an element attribute for each computation window. Its state contains the identifier of the current iteration, the definition elements to consider for each input block and the current sum value. It is important to notice that the definition of the block does not necessary match with the window definition. The aim is to be able to build the window result directly from block results.
This major distinction between stateless and stateful operators needs to be considered in the definition of a continuous query language.

2.4 Continuous query languages

We introduce three levels of continuous query languages. These languages provide tools to process data stream through different interfaces and with different language expressiveness.

CQL and SQL-derived languages. The first level of continuous query languages is based on SQL. Actually, languages like CQL[5], are built over SQL. They add to basic SQL implementation, the support of computation windows, described with more details in Section 2.2. In addition, they provide operators returning a stream from data streams or static data.
For example, CQL considers three classes of operators : Stream-to-Relation, Relation-to-Relation and Relation-to-Stream. Stream-to-Relation refers to computation window definition as illustrated in Section 2.2. These operators take as input a stream and return an *instantaneous relation* R according to the window definition. R is composed by tuples belonging to a window and can be processed as a static relation. Relation-to-Relation operators correspond to basic SQL operators (projection, selection, join...). Finally, Relation-to-Stream operators allow to build a stream from one or many instantaneous relations. CQL operators and optimization techniques are presented with more details in [5].

Box and arrow : graphical programming. Box and arrow paradigm [2] represents an application as a direct acyclic graph (DAG) of *boxes* connected by *arrows*. A box corresponds to an operator, stateless or stateful, taking stream elements as inputs. Arrows indicate how stream elements run between boxes. Considering a set ω of predefined operators including Stream-to-Relation, Relation-to-Relation and Relation-to-Stream operators, a continuous query can be defined graphically as a box and arrow DAG where boxes are composed of operators from ω. The main difference with CQL is that the continuous query can not benefit from automatic

query optimization based on algebraic properties. Performances of a box and arrow DAG depend more on user implementation than a graph generated from an equivalent CQL query.

Programming patterns. The expressiveness of pattern allow to define stateless and stateful operators. Thus, patterns take as input a stream element or a list of stream elements and apply a user-defined operator written in a programming language like JAVA, C or Python. There are two main differences between programming patterns and other continuous query languages. First, a continuous query must be defined as the composition of atomic operators. It increases deeply development and maintenance effort because each operator is user-defined. Second, the optimization of the global execution requires to rewrite operators one by one. No automatic optimization based on algebraic properties can be applied.

We have exposed notions and definitions necessary to handle data streams and define continuous queries. It is important to see then how these continuous queries are executed within a DSMS. For continuous query execution, two paradigms arise: workflow-designed queries and MapReduce framework.

2.5 Workflow

To clarify the concept of workflow, we remind the following definition.

Definition 5. (Workflow) *A workflow is a direct acyclic graph (DAG) where vertices are operators and edges define data transmission between operators.*

Independently from their semantics, operators can process data one by one, denoted *pipeline execution* (see Definition 6), or by block of data, for example aggregative operators. Operators communicate following two models: by invoking operator functions or via unbounded FIFO queues (pipeline) and buffers (block). In the context of stream processing, workflows present the advantage to be able to play on data and operator parallelism. Indeed, data streams can be processed in parallel without changing operator semantic. For example, while filtering important data stream volumes, data can be partitioned and distributed between multiple replicas of a single filter, each replica being processed independently. It is denoted as the *stream partitioning* pattern[19].

Definition 6. (Stream pipelining) *Let P be an operator which can be divided into k consecutive subtasks. Each P_i, $i \in [1;k]$, is denoted the i-th stage of P and is executed on an exclusive process.*

According to Definition 6, operators of a workflow can be seen as stages of a super operator. Data stream elements run then through all the stages sequentially. The limitation to stream pipelining is the presence of an aggregate operator requiring a set of data to compute a result.

Definition 7. (Stream partitioning) *Let P_1, P_2,..., P_k be k independent operators. They all take as input a subset of outputs produce by an operator P_0. In order to process the k independent operators in parallel, P_0 can split its outputs in k partition and distribute a partition to each P_i, with $1 \leq i \leq k$.*

According to Definition 7, an operator of a workflow can split its outputs to distribute partitions among multiple operators. The partition method varies according to the semantic of next operators. For example, an operator A distributing its outputs to k replicas of a same logic operator B. In that case, A can split its outputs only considering load balancing between replicas of B. But, if A is distributing its outputs to distinct and independent operators B and C, the partition policy depends on B and C semantic. Finally, a solution is that A replicate all its outputs for each following operator according to a workflow.

2.6 MapReduce paradigm

MapReduce [12] is a well-known framework developed initially to process huge amount of disk-based data on large clusters. The strength of this framework is to offer great parallelism with a simple programming paradigm. Actually, the core of any MapReduce application relies on two functions: Map and Reduce. These generic functions are defined as follow according to [12]:

$$\textbf{Map}\ (\text{key}_{in}, \text{val}_{in}) \longrightarrow \text{list}(\text{key}_{inter}, \text{val}_{inter})$$
$$\textbf{Reduce}\ (\text{key}_{inter}, \text{list}(\text{val}_{inter})) \longrightarrow \text{list}(\text{val}_{out})$$

As mentioned above, MapReduce framework aims disk-based data processing. Contrary to DBMS, MapReduce-based systems do not rely on data model to optimize treatments. In order to distribute great amount of data on a large cluster, data are partitioned with regards to cluster configuration (*e.g.* number of nodes executing Map and Reduce functions). Each partition is identified with a key used to affect the partition to a Map node. The scheduling between partitions and Map nodes follows distribution strategies like *Round-Robin* in order to balance computation load as good as possible. Each Map node executes the user-defined Map function on one or many partitions. The function produces a list of intermediate key/value list pairs depending on partition contents. Map phase outputs are then shuffled and sorted in order to make the Reduce phase easier. An optional phase, called *Combine*, can be processed on each Map node. The Combine phase consists in applying the Reduce function on Map outputs in order to have results for each partition. It may be useful while having potentially several redundant computation like in [8]. Each Reduce node gathers intermediate key/value list pairs and computes a list of value which are final results.

3 Related works

Previous surveys[7, 22] presents some workflow-based DSMS like Aurora[2], STREAM[4] and TelegraphCQ[10] appeared in order to deal with these new issues. Main objectives are to provide:

- Continuous query definition on data streams in a high level language including windowing schema support.
- A query execution architecture that produces results as fast as possible and minimize number of computation thanks to result mutualization.
- Structures to avoid that input rates overwhelm the query processor.

After works described in the survey[7], the MapReduce framework appears as a robust solution that scales up easily for highly parallel batch processing. Some solutions based on MapReduce have emerged [8, 24, 3, 15]. An other survey[14] presents some patterns related to ability of a DSMS to dynamically adapt their treatments to their execution environment. Those patterns are merged behind the notion of the *elastic stream processing*. In addition, a survey[19] exposes patterns and infrastructures dealing with failover management and treatment parallelization. It is relevant to present those issues in order to offer a global overview of stream processing deadlocks.
In this context, we suggest, through this survey, an up-to-date overview of stream processing techniques and management systems covering and comparing workflow-based and MapReduce-based DSMS.
.

4 Classification of stream processing systems

This section aims at facilitating the comparison of recent parallel and/or distributed DSMS according to some performance features. Our classification is based on three criteria of comparison. The first criterion concerns the **paradigm** of the solution, *i.e.*, the topology of a continuous query within a DSMS. Facing our constraints of parallelism and distribution, two paradigm are found in the literature: the former, named workflow-based, consists in turning a continuous query into a workflow (see Section 2.5) and distribute operators among nodes while the latter, named Map Reduce-based, consists in exploiting the Map Reduce paradigm to massively parallelize query processing. Hybrid approaches will be also considered. The second criterion allows to separate systems supporting **window incremental processing** or not. As presented in Section 2.2, window iterations can share panes. Panes size can be determined directly from window specifications. A DSMS can take advantage of this knowledge to compute results on panes and store them for mutualization between consecutive iterations of a sliding window. It aims at reducing computations only relying on an appriopriate stream discretization. Moreover, it allows to process in parallel consecutive panes and merge results after. In order to discretize streams according to panes, it requires that a DSMS includes window-oriented strategy and identification management at least for data scheduling or, in addition, for stream

acquisition. Moreover, pane management within a DSMS can open to other window-based improvements. We consider that a DSMS unable to process windows incrementally, includes **window batch processing** mechanisms. The last criterion includes the **support of the parallel execution** and allows to distinguish centralized multi-core solutions from distributed ones. Indeed, It appeared, like for batch processing, that the exploitation of a cluster of machines becomes necessary to scale up DSMS applications.

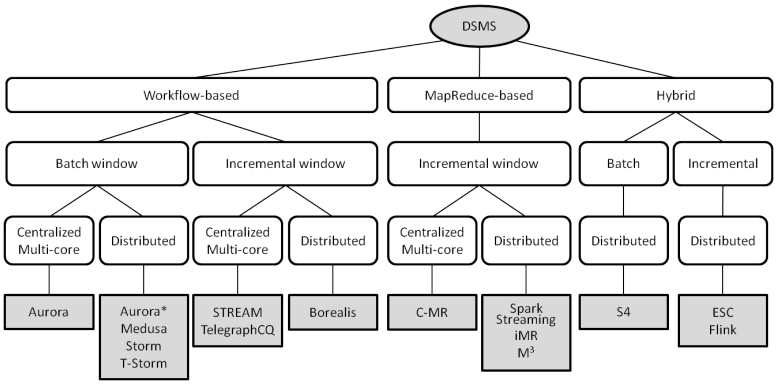


Fig. 2: Parallel Data Stream Management Systems classification

Figure 2 depicts our classification of different parallel stream processing techniques according to the previous criteria. For instance, the *Borealis* system is classified as a workflow-based DSMS computing window iteration results incrementally with the possibility to distribute query processing on a cluster. Moreover an extended classification considering other types of query (*i.e.* no continuous query) and other levels of data granularity is available here[6]. The rest of this section provides details for each solution classified according to our criteria.

4.1 Workflow-based solutions

Workflow-based solutions are chronologically the first to be developed. We suggest, for the remainder of this section, a generic runtime architecture. It is composed of three layers. The acquisition layer is an interface between inputs (raw data streams) and the query engine that execute the workflow. This layer includes acquisition units able to operate basic treatments on input streams like data fusion and data partitioning or

[6] http://liris.cnrs.fr/roland.kotto-kombi/PDSMS/classification/

complex treatments like *load shedding*. Load shedding aims at absorbing input stream variations in order that the processing layer respects latency and quality constraints.

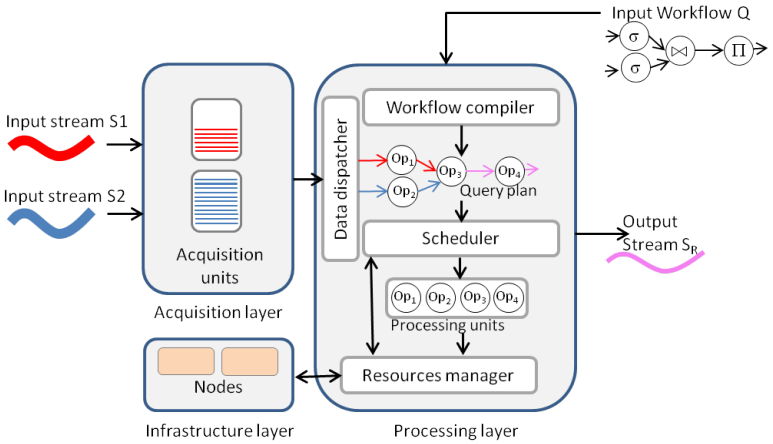


Fig. 3: Generic and global architecture for workflow-based DSMS

The processing layer is composed of five components. First, a workflow compiler turns a query or a workflow into an executable *query plan*. Second, a data dispatcher routes data to the appropriate operators. Then, a scheduler applies a strategy to distribute operators on processing units. Scheduling strategies could be based on operators cost, selectivity or inter-unit traffic. Next, operators are allocated on a processing unit in order to be executed. They potentially belong to multiple query plans. These operators are executed on physical computation nodes on the infrastructure layer thanks to a resource manager.

Window batch processing Firstly, we consider a centralized multi-core solution named **Aurora**[2]. Even if this DSMS includes window support thanks to an *Aggregate* operator[2], it does not compute results on panes but on complete windows. The lack of pane management prevents it to mutualize computations within a workflow. Moreover, considering time-based windows, the size of a window may vary deeply and require disk storage before processing. To tackle this issue, a timeout can be applied to the Aggregate operator. It avoids out-of-memory errors and guaranties a theoric maximal end-to-end latency. Aurora is a workflow-based solution running on centralized multi-core architecture. The objective of Aurora is to suggest a data stream-oriented solution that is not based on existing batch-oriented management systems. If we consider the Figure 3 anew, the acquisition units are implemented by queues with

priority policies. The scheduling strategy is based on *Quality-of-Service* (QoS) specifications defined by the user (e.g., latency). In Aurora, a continuous query corresponds to a workflow. This workflow is denoted an *application* and is composed by *operator boxes*. Boxes are linked by *arrows* to define how data streams run into the application. Aurora provides not only graphical tools to define applications, but also a syntax to write continuous queries in a SQL-like language based on a set of predefined primitives. Moreover, these primitives implement stateless and stateful operators. To execute stateful operators, some additional buffers are used in the processing layer. Aurora aims applications like sensor network monitoring. In fact, Aurora is limited to *small scale* applications because of the capacities of memory and cores.

Amongst distributed solutions that are tuple-oriented and based on workflows, we have identified some Aurora extensions [11, 9] and some Apache solutions [23]. As direct extension of Aurora project, **Aurora*** [11] aims at extending Aurora architecture with a computation cluster composed by heterogeneous machines. All machines must be under a same administration domain. Indeed, a master has to be able to gather information about each machine and distribute application fragments between machines. Aurora* applications can be write exactly like Aurora applications. Operator distribution is completely manage by the master's scheduler. It limits tuning opportunities to manage specifically operators. The gain obtained through distribution allows to target medium and large scale stream-based applications. Targeted applications are network monitoring, location tracking services or fabrication line management.
Aurora* installed main tools to distribute Aurora's applications on a cluster of machines. Nevertheless, many stream-based applications can be processed on clusters which are physically separated and grouped under administrative entities. These administrative entities encapsulate information on cluster nodes. The DSMS **Medusa**[9] aims then at providing an infrastructure to support federated operation across administrative boundaries. Actually, the master does not have information about each machine. The global cluster is composed of *participant*, themselves composed of a variable number of heterogeneous machines. Medusa distributes application fragments between participants. Each participant has, at the initialization, a same amount of *economical resources*. They can use those resources to delegate operators to an other participant in case of potential or effective overload. The global system follows microeconomic concept to reach a stability. Medusa relies on Aurora interface to define continuous queries. As Aurora*, Medusa targets medium and large scale stream-based applications.
Apache Storm[7] aims at providing a framework for scalable and fault-tolerant stream processing. It relies on a cluster of heterogeneous machines. According to Storm architecture, a machine, called a *worker node* is composed of *workers*. Each worker contains a variable number of *slots*. A slot is an available computation unit which has dedicated CPU resources. When an operator is affected to a slot, it is encapsulated in

[7] https://storm.apache.org/

an *executor*. According to our generic architecture, processing units are equivalent to workers. Each operator corresponds to an executor. Contrary to Aurora's architecture, acquisition units are explicitly operators of workflows in Storm. Continuous queries are represented as user-defined workflows, denoted as *topologies*. These topologies are also workflows but vertices belongs to two main categories: *Spout* and *Bolt*. Spouts are data transmission nodes and can be conceptualized as multiplexers. They provide an interface between sources and processing environment. After getting connected to one or many sources, they transmit data streams to one or many Bolts. Each Bolt execute a user-defined operator which is considered as atomic. Storm does not include primitives but provides programming patterns (see Section 2.4). Basically, Storm does not support stateful operators, so naturally does not support window incremental processing, but they can be added through API extension. Allocation of executors on workers is achieved by a scheduler minimizing CPU and memory usage. Storm targets applications handling huge volume of data like social data management.

T-Storm [23] extends Storm by suggesting a traffic-aware scheduler to handle potential network bottleneck. Topologies tends to be split in more dense subgraphs before operator allocations on slots.

Window incremental processing Some centralized solutions [4, 10] have appeared including window-based techniques due to the need to mutualize some calculus during sliding windows computations.

STREAM[4] aims at suggesting a data stream-oriented solution supporting the declarative and expressive SQL-like query language CQL. On the architecture level, STREAM differs from Aurora on two major aspects: i) the acquisition units exploit the *Stream-to-Relation* operators defined in CQL (see Section 2.4) and ii) the scheduler module is based on the *chain scheduling* algorithm minimizing memory usage during runtime[6]. Thanks to CQL, STREAM can automatically optimize SQL-like queries and turn them into query plan as explained in Section 2.5. As Aurora, STREAM supports stateless and stateful operators. Instead dedicating a core for a workflow, STREAM can identify operators common to multiple workflows and mutualize computations. But, like Aurora, it aims small scale applications. Targeted applications by STREAM deal with real-time monitoring like financial analysis.

TelegraphCQ[10] aims at offering a flexible infrastructure for fault-tolerant stream processing. TelegraphCQ totally differs from the generic architecture. Acquisition units are replaced by ingress and caching *modules* like a sensor proxy to create an interface with external data sources. TelegraphCQ engine, called *Eddy*, replaces processing units and considers a workflow only like an operator set with a routing policy. Operators are connected to the *Eddy* and all operator inputs and outputs pass through him. TelegraphCQ provides an hybrid interface for continuous query definition. Actually, it offers a set of predefined operators (project, join, filter, tumble...). But these operators can only be combined for the definition of operators respecting programming patterns (see Section 2.4). A

TelegraphCQ application is then a workflow of user-defined stateless and stateful operators and each operator is the composition of primitive. Like Aurora and STREAM, it can handle small scale applications because the star-like model, centred on the Eddy, may generate an important inter-operator traffic increasing deeply with data stream volumes. Targeted applications are event-based business processing.

Amongst existing distributed workflow-based solutions, we identify **Borealis**[1], solution derived from Aurora*, as a distributed solution which considers windows for performance improvement. The main objective of Borealis is to improve result quality by softening the impact of real-world stream defects during processing. Indeed, some anomalies may be due to not only an inefficient tuple transport inside an overloaded network but also the emission of incorrect tuple values. Corrections can be done by dynamically revising query results and modifying query specifications (*e.g.*, modifying latency between two consecutive results). Taking advantage of Aurora* and Medusa improvements, Borealis can be executed under one or multiple administrative domains. As Aurora* and Medusa Borealis is based on Aurora's graphical interface and syntax for continuous query definition. Borealis aims more specifically applications based on self-correcting stream sources like financial service applications.

	Aurora	Aurora*	Medusa	Borealis	STREAM	Storm	T-Storm	TelegraphCQ
Execution support	centralized multi-core	distributed	distributed	distributed	centralized multi-core	distributed	distributed	centralized multi-core
Continuous query definition	graphical	graphical	graphical	graphical	CQL or graphical	API	API	API
Workflow terminology	application	application	application	application	query plan	topology	topology	query plan
Vertices terminology	boxes	boxes	boxes	boxes	operators	spouts or bolts	spouts or bolts	modules
Primitive operators	yes	yes	yes	yes	yes	no	no	yes
Incremental window support	no	no	no	yes	yes	no	no	yes
Operator scheduling	QoS-based	QoS-based	Contract-based	QoS-based	CPU and memory based	CPU and memory based	CPU, memory and network traffic based	CPU and memory based
Failover management	no	yes	yes	yes	no	yes	yes	no
Quality evaluation	yes	yes	yes	yes	yes	no	no	no
Quality definition scope	workflow	execution node	execution node	vertex	vertex	-	-	-
Application example	sensor monitoring	network monitoring	network monitoring	stock market analysis	financial analysis	stock market analysis	stock market analysis	event-based business processing

Fig. 4: Workflow-based DSMS features

To sum-up (see Figure 4), workflow-based DSMS can process sets of stateless and stateful operators on data streams[2, 4]. The definition of a workflow can be done following two ways. On one hand, a workflow is derived from a global continuous query[4]. These DSMS benefit from algebraic optimization. On the other hand, a workflow is defined operator per operator. Each operator is composed of predefined operators[2, 10] or through an API[23]. They all take advantage of stream pipelining but there are different scopes : multi-core on a single machine [2, 4] or distributed on a cluster of heterogeneous[11, 9, 1, 23]. This scope impacts the scalability of a DSMS.

4.2 MapReduce-based solutions

MapReduce-based DSMS [8, 16, 24, 3] are all designed to support windowing schema and they can not return results as soon as data arrives in the system. They must consider finite substreams which corresponds to window definition. Nevertheless, it is rarely relevant to process an entire window at a single time because it could represent huge volume of data and delay the next window treatment. In this context, an other window-oriented solution appeared to tackle stream discretization issue. We suggest to separate MapReduce-based DSMS in two categories: *Pipeline* DSMS [8, 16] which executes asynchronously Map and Reduce phase as soon as they receive new inputs, and *Buffer-based* DSMS [24, 3] which collects data pane-per-pane and computes Map and Reduce phase after. Moreover, it is worth noting stream elements are turned into key/value pairs to fit with MapReduce framework. Timestamps are neither key nor value but only metadata to support windowing schema. Most phases described in Section 2.6 are also implemented with some variations. The main difference with batch-oriented systems like Hadoop is that sources are data streams. Beyond the obvious acquisition challenge data streams represents, it is also important to notice that handling multiple sources is difficult. For some systems [8, 3], handling multiple sources is solved by routing elements considering their respective keys.

Pipeline systems rely on an asynchronous execution of Map and Reduce phase. A pipeline DSMS based on MapReduce, called *Continuous-MapReduce* (C-MR) suggests an implementation of asynchronous MapReduce. The aim is to take advantage of stream partitioning (see Definition 7) based on key value. In **C-MR** [8], data streams are infinite sequence of key/value pairs. As soon as a new element arrives in the system, it triggers a Map operation. A node can then execute a Map operation without considering which operation is running on other nodes. Elements are routed to Mappers considering their respective keys. A specificity of C-MR is that the Combine phase, described in Section 2.6, is mandatory. Thanks to the Combine phase, C-MR is able to generate a pane oriented scheduling. Indeed, Map phase outputs are sorted according to their timestamps and gathered on a node executing a Reduce function on a pane content. To materialize the end of a pane during the execution, punctuation mechanisms are used. Each source sends a specific tuple which marks that all tuples for a given pane have been sent. When

a Combine node receives punctuation tuples from all sources, the execution starts. C-MR essentially exploits these Combine nodes to avoid redundant computations between consecutive sliding windows. An other solution based on Hadoop[15] aims to take advantage of stream pipelining between mappers and reducers. The main contribution relies on a scheduler based on hash functions optimizing jobs.

Some buffer-based solutions have been developed. The objective of those DSMS is to discretize data streams and process each batch as any disk-based data. Nevertheless, a complete MapReduce job is not triggered from scratch for each batch. Apache **Spark Streaming**[24] brings the *Apache Spark Engine* [8] to stream processing. Apache Spark is a parallel engine which executes DAG obtained from a SQL-derived query or a MapReduce job. A graph is decomposed in MapReduce jobs by Apache Spark and its execution is optimized to fit on main memory. Spark Streaming appears then as an interface receiving tuples from data streams and discretizing them into multiple batches. It directly corresponds to our acquisition layer. Each batch is then processed by the Spark Engine. According to Spark Streaming terminology, data streams are turned into *Discretized Stream*, or *DStream*. A DStream is a potentially infinite sequence of *Resilient Distributed Dataset* (RDD). A RDD is defined by a timestamp range and it is important to notice that this range is equivalent for all RDDs. A RDD can be considered as a pane (see Figure 5) explicitly defined by a user. Spark Streaming supports the definition of MapReduce jobs on sliding windows. As window size is a whole number of RDDs, Spark Streaming can foresee which RDDs are involved in multiple window computations. They are then cached as long as a window benefits the intermediate result. It belongs to the second generation of stream processing engines for large scale applications like social network data management.

The motivation of iMR [16] is that many log processing applications produce much less data than they consume. In this context, it is relevant to process data streams locally and then send results for storage. Runtime architecture of iMR is similar to C-MR's architecture except operator granularity. The aim remains to group pane results for potential multiple reuses. But an important distinction is that iMR triggers a Map/Reduce operation for a list of elements. In addition, iMR suggests an *uncombine* operator to allow incremental operations. The physical implementation of iMR relies on a cluster of machines. The heterogeneity of machines is not handled by iMR's resources manager.

$\mathbf{M}^3$[3] is an implementation for MapReduce execution exclusively on main memory for stream processing. The objective is to suggest a MapReduce DSMS resilient to input rate variations by dynamically revising discretization parameters. Instead of fixing the acquisition buffer size, M^3 is based on a dynamic load balancing mechanism between Map nodes. In fact, stream discretization aims at processing approximately the same amount of data instead of triggering an execution at fixed timestamps.

[8] https://spark.apache.org/

	C-MR	iMR	Spark Streaming	M^3
Execution support	centralized multi-core	distributed	distributed	distributed
Continuous query definition	API (MapReduce job)	API (MapReduce job)	SparkSQL or API (MapReduce job)	API (MapReduce job)
Window management	punctuation	buffer-based	buffer-based	buffer-based
Pane terminology	pane	pane	RDD	pane
Operator scheduling	window-aware	window-aware	window-aware	window-aware
Failover management	no	no	yes	yes
Application example	sensor monitoring	logs analytic	social data analysis	network monitoring

Fig. 5: MapReduce DSMS features

To summarize (see Figure 5), MapReduce-based DSMS integrate window-oriented schedulers. The objective is to obtain intermediate results on windows or subwindows for reuse. They take advantage of stream partitioning (see Defintion 7) in order to parallelize computations. In comparison with workflow-based DSMS, the definition of continuous queries can be done operator per operator through MapReduce APIs [8, 16, 3] but also globally with a SQL-derived [24].

4.3 Hybrid solutions

Hybrid DSMS rely on continuous queries represented by a workflow as defined in Aurora*. But contrary to workflow-based DSMS, data are turned into *events* [17]. An event is a key/value like MapReduce inputs. The key represents the type of the data and the value, denoted the *attribute*, corresponds to the tuple of the stream.

Window batch processing S4 [17] is an hybrid DSMS without window-based optimization. It enriches workflow representation. Indeed, each operator is associated to two metadata: a type list of consumed events and a value list for each of them. It forms a *Processing Element* according to S4 terminology. Each Processing Element has a dedicated buffer which contains data to process. Data are grouped by key to make operator execution easier like Reduce operations. Operators are user defined but must respect patterns to guaranty that their execution can be done in parallel. Even if S4 benefits from both paradigms, it lacks many

features. Contrary to most workflow-based DSMS, S4 does not support a continuous query language. It compels developers to define *ad hoc* operators for each query. Moreover, queries must be designed as a set of Processing Element operators so difficult to translate for other DSMS. It does not support windowing schema natively. Nevertheless, the integration of windowing schema only requires to set time-based buffers for relevant Processing Element, *e.g.* an aggregate operator. Finally, S4 defines static routes between operators according to the global workflow.

Window incremental processing Based on a similar hybrid architecture than S4 but including window-based scheduling strategies, **ESC** [20] is a DSMS aiming real-time stream analysis like pattern mining. ESC has been developed as a platform for Cloud computing. The execution support must be a cluster of homogeneous virtual machines in terms of performances. It includes CPU speed, main memory capacity and also bandwidth. Like S4, ESC represents a query as a DAG where each vertex is a Processing Element. ESC offers more flexibility than S4 because input Processing Elements are similar to Apache Storm Spouts [23]. Actually, they do not execute a specific task on data but only get connect to multiple sources and send data to other Processing Elements which execute operators specified by the workflow. Data are represented as events different from S4 ones. ESC events are sets of four elements: a type, a context, a key and an associated value. The key/value pair is exploited as it is in S4 architecture. The type is the semantic representation of an event. For example while processing stock market data, the type can be "stock value" or "stock evolution" and the key, the acronym representing a given stock. The context corresponds to the query interested by the event. Comparing to S4, ESC suffers less drawbacks because window support is effective through time-based buffers. In fact, tumbling windows are supporting thanks to a tick mechanism. ESC includes a clock which emits a tick at regular interval. When a tuple arrives in the system, it is affected to the closest tick which is used as timestamp. In addition, all operators are synchronized and must process data belonging to their respective buffers and flush them after. Nevertheless, ESC only provides function patterns to define operators. **Apache Flink** is a DSMS which relies on a distributed streaming dataflow engine. As Storm, Flink allows the definition of a continuous query as a workflow of operators sending their outputs to one or many other operators. Nevertheless, Flink supports a SQL-like language enriched with the operators Map and Reduce. In this way, Flink is clearly an hybrid DSMS exploiting both stream pipelining and stream partitioning. Finally, Flink implements a specific memory management to tackle garbage collection issues.

We have exposed several DSMS according to our classification and it emerges that stream processing on tumbling or sliding windows suffers many constraints. Workflow-based solutions [2, 10, 9] aims at providing results as soon as possible for monitoring and real-time analysis applications. They are mostly expressive [4] through the definition of a continuous query language [5] which includes stateful operators and ex-

plicit windowing schema support. Comparing to Workflow-based DSMS, MapReduce-based DSMS [8, 24, 16, 3] are designed to store and reuse intermediate results. Our classification relies on logical criteria like the integration of window-based optimization, conceptual like the paradigm and physical like the execution support. Nevertheless, aspects dealing with the adaptation of DSMS to their environment of execution must also be considered for a complete overview.

5 Complementary issues for stream processing

This section aims at covering most aspects for resilient DSMS design. In a first time, we introduce some dynamic optimization techniques for *elastic stream processing* used in DSMS presented in Section 4. Secondly, failover issues are presented. Some existing solutions are presented with their limits. Then we introduce *Quality-of-Service* evaluation and different variants existing in some DSMS.

5.1 Elastic stream processing

As introduced in Section 2.1, data streams are unpredictable. Input rates may vary deeply in terms of volume and distribution of values at any moment during runtime. It may lead to bottlenecks at the acquisition and processing layers. It is then an important aspect to consider in the analysis of a DSMS in order to estimate his availability during execution. Dynamic reconfiguration patterns, gathered under the notion of *elastic stream processing*[14], are exploited by DSMS to tackle this issue.
Spark Streaming[24] does not integrate auto-reconfiguration mechanisms because of its architecture. Actually, Spark Streaming sets its acquisition units to process RDD per RDD (see Figure 5) and bases mutualization of computation on this strategy. Nevertheless, tuning opportunities are provided like the configuration of the level of task parallelism. Spark Streaming also allows to defined each RDD size on its memory size in order to avoid triggering execution for a low amount of data. S4[17] decentralized architecture prevents global reconfiguration of the workflow. Processing units are managed locally and workflow edges are static. However, load balancing is managed locally by each processing nodes. As reminder, S4 operators, denoted processing elements, are grouped by processing nodes. A processing node belongs to a single physical machine. Instead managing load balancing, iMR[16] is able to apply adaptive load shedding policies on each processing unit to balance input rate increases. The architecture of M^3 allows to dynamically adapt acquisition units, more precisely buffer sizes, to gathered multiple panes in a buffer or acquire a pane on multiple buffers. C-MR[8] includes more sophisticated elastic mechanisms. Scheduling strategies can be enabled to decrease as much as possible global latency (*Oldest-Data-First* strategy) or decrease memory usage to avoid out-of-bound errors (*Memory-Conservative* strategy). Storm bases *operator sliding* more on resources required by an executor (see Section 4). The objective is to balance load among worker nodes. As

described in Section 4, T-Storm modifies Storm's scheduler by considering in priority inter-worker traffic generated by an execution layout. But those strategies do not take advantage of *operator reordering* [14]. Actually, some operators can be compute without changing final outputs, *e.g.* commutation of filters. Operator reordering is exploited by ESC[20]. The master process analyses, during runtime, costs of operators, and modifies execution order if necessary. In the same way, STREAM[4] takes advantage of SQL support to operate a dynamic algebraic optimization. STREAM optimizer is based on operator selectivity and average end-to-end latencies. Aurora[2] suggests many elastic mechanisms even it can not take advantage of a cluster of machines.*Operator boxes* (see Section 4) can be combined to be executed in a single time. For example, a projection of some attributes followed by a filter can be processed in a tuple-per-tuple on a single node. Moreover, projections can be automatically added in an application to decrease tuple size without loosing relevant attributes. Aurora also integrates operator reordering like ESC and load shedding. Finally, Aurora is able to dynamically change some operator implementations like joins. *Algorithm selection* [14] requires to monitor each box end-to-end latency in order to evaluate if an other implementation can process data more efficiently. Aurora's extents (Aurora*[11], Medusa[9], Borealis[1]) includes all Aurora's elastic mechanisms and implements to mechanisms for load balancing among nodes : *operator sliding* and *operator split*[11]. An operator slides from an execution node to an other to avoid local overload. Operator split aims at taking advantage of data partitioning to parallelize the execution of an operator. Finally, TelegraphCQ[10] integrates operator reodering thanks to its centralized processing engine Eddy[10].

5.2 Failover management

Failover management aims at balancing dynamically execution environment variations which decrease safety of the system. Contrary to elastic stream processing, safe failover induces resource consumption at any time during the execution to prevent quality degradation because of a failure. Considered failures are complete node failures (neither CPU nor memory are available) implying data loss. The aim is then to prevent those losses through three failover patterns [19] for safety improvement: *simple standby*, *checkpointing* and *hot standby*.
DSMS supporting only a centralized multi-core architecture (Aurora, TelegraphCQ, C-MR) do not integrate failover management. Some DSMS make a compromise between end-to-end latency and failover management. Indeed, S4 and iMR accept lossy processing. If iMR provides an explicit feedback on data losses, S4 only restarts a new node with lost operators on current data. M^3 and Spark Streaming rely on a cluster of Zookeeper[9] nodes to operate checkpointing[19] and restart failed operators on one or many active nodes. The distinction between Spark Streaming and M^3 is that M^3 stores operator states on main memory exclusively. Storm and T-Storm use a heartbeat mechanism to detect node

[9] https://zookeeper.apache.org/

failures. Actually, the master node receives continuously heartbeats from worker nodes and try to restart them if a heartbeat is not received for a predefined timeout. Operator states are stored on a shared memory of a Zookeeper cluster or on disk. Aurora's extents (Aurora*, Medusa and Borealis) guaranty *k-safety*. The failure of any node k does not impact the final result of an application. In order to offer that guarantee, these DSMS discard tuples lazily. A tuple is lazily discarded if it is deleted only after it does not serve as input to any operator of a workflow.

5.3 Quality evaluation

Stream processing implies to process continuous queries on potentially infinite data. As exposed above, execution environment may induce irreversible data loss. Some DSMS [2, 1, 16] aim at providing a feedback to end-users on data losses. This feedback is conceptualized as a *quality* score. The quality is used as a threshold to make the difference between satisfying and unsatisfying results. Aurora [2] integrates quality score included in a *Quality-of-Service* (Qos) specification. QoS is defined as a function depending on performance or result accuracy expectancy. Users do not define a threshold value for quality but define which parameter to give advantage. Aurora application tends then to maximize final results' quality scores. In order to control quality, Aurora resource management is based on QoS-aware algorithms.
Nevertheless, Aurora is designed for centralized multi-core execution and QoS-aware algorithms are not adapted for distributed architectures. Indeed, QoS definitions used in Aurora do not consider that data can be lost because of network failures. In order to deal with this issue, Aurora* is able to infer intermediate QoS constraints for any node of a Aurora* cluster. Intermediate QoS constraints are inferred only for *internal* nodes. They are nodes which are neither connected to sources nor final outputs. Borealis [1] extends QoS specification to a more fine-grained level. Actually, Aurora* is able to infer QoS specification for each node output, each node executing a subgraph of the global Aurora application. But there is no inner-node QoS control from user's side. In this way, Borealis[1] allows to define QoS specification for any vertex in the dataflow (see Figure 4).
An other quality function, denoted C^2 [16], aims at providing information about the *completeness* to end-users. Actually, C^2 is defined as a spatio-temporal quality score. The spatial aspect represents resource consumption involved by the computation of the current result. The temporal component is more related to Aurora's QoS and provides information about data loss during last window computation.

6 Conclusion

In this paper, we have proposed a classification of Data Steam Management Systems according to their paradigm, their capacities to handle windowing and the type of infrastructure on which they can be deployed. To offer to readers an additional point of view about those DSMS we

consider some aspects related to elastic stream processing, failover management and the evaluation of the quality of results. It appears that targeted applications are the main key to decide which DSMS will more likely deliver best performances. In the case of a an application composed by complex operators handling potentially important volumes of data, updating results as soon as possible and with a tolerance to non optimal accuracy, Borealis appears as the best choice. It includes high level operators has automatic mechanisms for workflow optimization. Moreover, it supports window incremental processing and QoS specifications. Storm delivers great performances in a similar context but development and maintenance efforts are important. In an other case, an application computing periodically results that can be totally or partially reused on potentially huge volume of data will be handled efficiently by Spark Streaming. The use of Discretized Streams [24] allows mutualization of intermediate results and tuning opportunities soften elastic stream processing issues. This survey underline that in the era of Big Data and GreenIT, no solution are completely adapted and full satisfying. Indeed, existing systems suffer from not having specific optimization at each steps of the query processing.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[3] A.M. Aly, A. Sallam, B.M. Gnanasekaran, L. Nguyen-Dinh, W.G. Aref, M. Ouzzani, and A. Ghafoor. M3: Stream processing on main-memory mapreduce. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1253–1256, April 2012.

[4] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Springer, 2004.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, VLDB Journal, 2003.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *ACM International Conference on Management of Data (SIGMOD 2003)*, 2003. An extended version of this paper titled "Operator Scheduling in Data Stream Systems" appears on this publications server as technical report 2003-68 at http://dbpubs.stanford.edu/pub/2003-68. This technical report proves an NP-completeness result showing the intractability of the problem of minimizing memory. The report also contains theoretical results and experiments for miminizing run-time memory requirements subject to user-specified latency constraints.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[8] Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, and Ugur Cetintemel. C-mr: Continuously executing mapreduce workflows on multi-core processors. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 1–8, New York, NY, USA, 2012. ACM.

[9] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load management and high availability in the medusa distributed stream processing engine. In *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 929–930, 2004.

[10] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[11] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[13] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[15] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.

[16] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, Dec 2010.

[18] A. Rosenthal, U. S. Chakravarthy, B. Blaustein, and J. Blakely. Situation monitoring for active databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 455–464, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[19] Kai-Uwe Sattler and Felix Beier. Towards elastic stream processing: Patterns and infrastructure. In Graham Cormode, Ke Yi, Antonios Deligiannakis, and Minos N. Garofalakis, editors, *BD3@VLDB*, volume 1018 of *CEUR Workshop Proceedings*, pages 49–54. CEUR-WS.org, 2013.

[20] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Com-*

puting (CLOUD), 2011 IEEE International Conference on, pages 348–355, July 2011.

[21] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 469–478, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[22] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[23] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544, June 2014.

[24] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: A fault-tolerant model for scalable stream processing. Technical Report UCB/EECS-2012-259, EECS Department, University of California, Berkeley, Dec 2012.