Affetti *et al. J Big Data (2017) 4:12*
DOI 10.1186/s40537-017-0072-9

Journal of Big Data

SURVEY PAPER Open Access

Defining the execution semantics of stream processing engines

Lorenzo Affetti*, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola and Emanuele Della Valle

*Correspondence:
lorenzo.affetti@polimi.it
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, DEIB, Milano, Italy

Abstract

The ability to process large volumes of data on the fly, as soon as they become available, is a fundamental requirement in today's information systems. Modern distributed stream processing engines (SPEs) address this requirement and provide low-latency and high-throughput data stream processing in cluster platforms, offering high-level programming interfaces that abstract from low-level details such as data distribution and hardware failures. The last decade saw a rapid increase in the number of available SPEs. However, each SPE defines its own processing model and standardized execution semantics have not emerged yet. This paper tackles this problem and analyzes the execution semantics of some widely adopted modern SPEs, namely Flink, Storm, Spark Streaming, Google Dataflow, and Azure Stream Analytics. We specifically target the notions of windowing and time, traditionally considered the key distinguishing factors that characterize the behavior of SPEs. We rely on the SECRET model, introduced in 2010 to analyze the windowing semantics for the SPEs available at that time. We show that SECRET models well some aspects of the behavior of modern SPEs, and we shed light on the evolution of SPEs after the introduction of SECRET by analyzing the elements that SECRET cannot fully capture. In this way, the paper contributes to the research in the area of stream processing by: (1) contrasting and comparing some widely used modern SPEs based on a formal model of their execution semantics; (2) discussing the evolution of SPEs since the introduction of the SECRET model; (3) suggesting promising research directions to direct further modeling efforts.

Keywords: Stream processing, Stream processing engines, Modeling, Execution semantics, SECRET, Time semantics, Windows

Introduction

Several modern data-intensive applications need to process large volumes of data on the fly as they are produced. Examples range from credit card fraud detection systems, which analyze massive streams of credit card transactions to identify suspicious patterns, to environmental monitoring applications that continuously analyze sensor data, to click stream analysis of Web sites that identify frequent patterns of interactions. More in general, stream processing is a central requirement in today's information systems.

This state of facts pushed the development of several stream processing engines (SPEs) that continuously analyze streams of data to produce new results as new elements enter the streams. Unfortunately, existing SPEs adopt different processing models and standardized execution semantics have not yet emerged. This severely hampers the usability

Springer Open

© The Author(s) 2017. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

and interoperability of SPEs, since a user needs to understand system-specific aspects to confront various alternatives and select the ones that better suite her needs.

The main factors that differentiate the behaviors of SPEs are the models of *windows* and *time* they adopt [1]. Windows enable computations that would be otherwise unfeasible on unbounded datasets such as streams. For instance, counting the number of elements in a stream is not possible, since the stream never terminates. The common solution consists in splitting the stream into finite blocks of contiguous elements—called windows—and performing the computation within the bounds of each window. Several types of windows exist: for instance, windows can be defined either based on the number of elements they contain or based on time boundaries, and they can partition a stream into non-overlapping chunks or contain common elements [1].

The semantics of windows strictly depend on the related concept of time, which determines how the incoming elements are associated to different windows. In some cases, time is seen as meta-data associated to each element in a stream either by the source that produces that element or by the SPE itself. In other cases, time is not associated to elements and the SPE refers to the system clock of the physical machine it is running on.

In 2010, the idea that the semantics of windows and time are central to the understanding of SPEs motivated the definition of SECRET, a model that captures the behavior of windows in the state-of-the-art SPEs of that time [2]. SECRET was used to analyze both academic and industrial SPEs. The semantics of the former were typically formally defined—yet different from system to system—, while the semantics of the latter were most often dependent on implementation details. SECRET could effectively capture and confront the behavior of all the systems it was applied to.

After the definition of SECRET, the increasing number and the growing complexity of real-time data analytics applications led to a bloom of new *distributed SPEs* that target cluster platforms to scale with the volume and velocity of input data. This poses two interesting and related research questions: (1) how did the semantics of windows and time change in distributed SPEs? (2) Is SECRET still adequate to fully capture the execution semantics of modern distributed SPEs?

This paper aims to answer such questions by using SECRET to model five distributed SPEs—Flink, Storm, Spark Streaming, Google Dataflow, and Azure Stream Analytics—that were developed after the introduction of SECRET and are today widely used in companies at the scale of Google, Twitter, and Netflix. We show that SECRET models well a subset of the behaviors of these systems and we shed light on the recent evolution of SPEs by analyzing the elements that SECRET cannot fully capture. To the best of our knowledge, this is the first work to investigate the adoption of a model that formally captures some aspects of the execution semantics of distributed SPEs.

The paper contributes to the research on SPEs in several ways: (1) it provides a precise modeling of five modern SPEs using SECRET; (2) it compares the systems and highlights their similarities and differences, thus helping the users to identify the systems that better satisfy their requirements; (3) it identifies some aspects of modern SPEs that SECRET cannot fully capture; (4) it discusses the evolution of SPEs since the definition of SECRET and suggests promising directions for future modeling efforts.

We release the entire code of the infrastructure we used for our analysis to enable the repeatability of the experiments and to ease the extension to other systems.

The remainder of this paper is organized as follows. "Background" presents background information on the key concepts of stream processing and on the SECRET model. "Analysis of stream processing engines" analyzes five widely used distributed SPEs using the SECRET model. "Discussion" discusses the insights that emerge from the modeling. "Related work" presents related work and "Conclusions" concludes the paper and presents future work.

Background

This section provides background information on "Stream processing engines" and on the "SECRET" model.

Stream processing engines

Stream processing engines (SPEs) provide abstractions to operate on dynamic datasets and produce new results on the fly as new input data becomes available. SPEs take in input one or more *streams*, which are append-only unbounded sequences of data, and produce output streams as a result of their computation.

The abstract architecture of an SPE is presented in Fig. 1. SPEs receive input streams from one or more sources—grey diamonds in Fig. 1—and organize the computation into a directed graph of operators—white circles and boxes in Fig. 1—either explicitly or implicitly. In the latter case, the developers are provided with high-level languages that are automatically translated by the SPE into the operator graph. Some systems adopt functional languages and provide composable functions—such as `map`, `filter`, `reduce`, etc.—that transform input streams into output streams. Other systems adopt declarative, SQL-like languages that represent the processing as queries that get repeatedly and continuously evaluated as new data becomes available. Modern SPEs take advantage of cluster architectures and deploy the operators in the graph on different cluster nodes, possibly replicating them. Organizing the computation into separate operators enables for task parallelism—different operators run on different threads on the same machine, or on different machines—, while replication enables for data parallelism—different portions of an input stream are processed in parallel on different instances of the same operator.

In this paper we separate operators in two classes: processing operators and windows. *Processing operators* apply a function to each and every element of their input streams. For instance, a filter operator selects or discards input elements based on a user-defined predicate, and a map operator converts each element of the input stream into an element

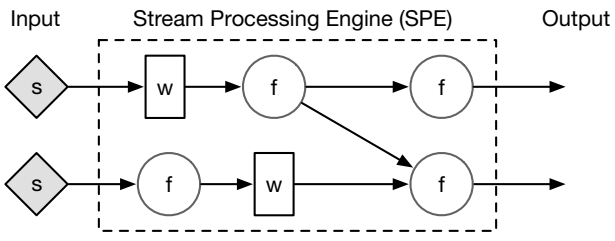


Fig. 1 The architecture of an SPE

of the output stream based on a user-defined function. We represent them in Fig. 1 as white circles.

Windows enable computations that would be otherwise impossible due to the unbounded nature of streams. For instance, the average value of a stream of integers could only be computed after reading the *entire* stream, which is impossible since the stream never terminates. Windows obviate this problem by isolating the portions of the input streams upon which the function embedded within an operator should be applied [1]. We represent them in Fig. 1 as white boxes.

Windows capture contiguous stream elements and are defined in terms of two parameters: a *size* that indicates the length of each window, and a *slide* that determines the interval between two consecutive windows. *Count windows* define the size and the slide based on number of elements. *Time windows* define the size and the slide based on some notion of order—*time*—between the elements in their input stream.

The behavior of time windows also depends on the notion of time, and different SPEs adopt different time semantics. Using the terminology of Akidau et al. [3], we say that an SPE adopts *processing time* semantics when each operator considers the clock time of the physical machine it is running on. In this case, elements do not have an attached timestamp and their mutual order is implicitly defined by the order in which they enter the operator. Processing time semantics leads to non-determinism, since the behavior of the system depends on the speed at which elements enter the engine and on the speed at which the elements flow between operators inside the engine, which is not under direct control of the developer. For these reasons, processing time has been often subject to critics, especially when considering distributed deployments of operators [3]. Nevertheless, some modern SPEs adopt processing time, under the implicit assumption that the clock skew between the physical machines where the SPE is running is negligible for the application at hand, and that the operators do not introduce significant delays.

We say that an SPE adopts *ingestion time* semantics when each input element is timestamped when it first enters the engine, assuming that a single machine receives and timestamps all the input elements. Each time-dependent operator refers to such timestamp during evaluation. Given a specific timing of arrival of input elements, ingestion time ensures that the engine produces deterministic results.

We say that an SPE adopts *event time* semantics when each element is timestamped by the source that produces that element, outside the SPE. In the presence of event time, operators might receive elements out of order, due to clock skew between sources or due to network latency. In this case, the operators temporarily buffer the input elements and reorder them before processing. If the engine waits for enough time to receive all out of order elements—that is, if no element is lost—then event time ensures determinism.

To better explain the notion of windows, Fig. 2 shows the effect of a time window of size 3 and slide 1 over an input stream. Stream elements are identified by upper case letters and ordered based on their timestamp—event time semantics. Notice that event time defines a partial order: for instance, elements `C` and `D` both have timestamp $t = 3$. Let us consider windows starting from $t \geq 1$. Window `w1` includes all the elements from $t = 1$ excluded to $t = 4$ included, that is `B`, `C`, and `D`. Window `w2` includes `C`, `D`, and `E`.

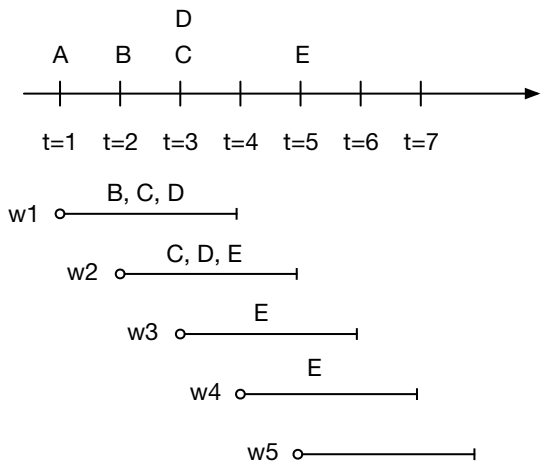


Fig. 2 Example of time window with size 3 and slide 1

Time windows can contain a heterogeneous number of elements, as in the case of `w1` and `w3`. Furthermore, they might be empty, as in the case of `w5`.

In general, the results produced by an SPE depend on (1) the topology of the operator graph; (2) the functions implemented in each operator; (3) the semantics of windows and time.

The first two elements are application-specific: they are defined by the developers and are not system dependent. Instead, the semantics of windows and time varies from system to system and thus is the key element to capture the differences between the available SPEs. The SECRET model we adopt in this paper focuses on the semantics of windows in the case of event time.

SECRET

SECRET [2] models both time windows and count windows and builds on the following assumptions: (1) stream elements have an associated timestamp—event time semantics—that defines a partial order between elements; (2) each operator reorders the input elements and processes them in timestamp order.

SECRET defines the semantics of windows by introducing the concepts (functions) of $\mathcal{S}$cope, $\mathcal{C}$ontent, $\mathcal{R}$eport, and $\mathcal{T}$ick. For ease of explanation, we present the SECRET formalism with reference to time windows, and we briefly discuss the differences in the case of count windows at the end of the section. The interested reader can refer to complete formalization of SECRET for further details [4].

As discussed in "Stream processing engines", each window operator is characterized by a size and a slide, and splits the input stream into windows all having the same size. SECRET defines windows as an interval $(t_o, t_c]$ where t_o is the start time (excluded from the window) and t_c is the end time (included in the window). A window is *open* at time t if $t_o < t \leq t_c$. A window is *closed* at time t if $t > t_c$. A window w contains an element e of an input stream if the timestamp of e is within the boundaries of w.

$\mathcal{S}$cope is a function that maps each point in time t to its *active window*, which is the open window w with the lowest t_o.[1] $\mathcal{S}$cope only depends on the size and slide of the window operator, and on the start time of very first window—t_0—, which is system specific.

$\mathcal{C}$ontent is a function that maps each point in time t to the stream elements that are in the active window at t.

$\mathcal{R}$eport is a function that defines the strategies that a window adopts to make its $\mathcal{C}$ ontent visible for downstream processing. A window can adopt any combination—conjunction—of the following four strategies: (1) *content change*: w reports only if the window content changes (with respect to the previous report); (2) *window close*: w reports only when the active window closes; (3) *non-empty content*: w reports only if the active window is not empty; (4) *periodic*: w reports only at regular time intervals.

With reference to Fig. 2, an SPE that reports on *content change* would not report the window `w4`, since its content is identical to the content of the previous window `w3`. Similarly, an SPE that reports on *non-empty content* would not report window `w5`, since it does not contain any element.

$\mathcal{T}$ick is a function that describes the conditions that trigger a possible $\mathcal{R}$eport. SECRET identifies the following alternative $\mathcal{T}$ick strategies: (1) *tuple-driven*: each incoming element triggers an evaluation; (2) *time-driven*: each time progress triggers an evaluation.

With reference to Fig. 2, an SPE that adopts a *tuple-driven* $\mathcal{T}$ick would evaluate the $\mathcal{R}$ eport conditions twice for time $t = 3$, since two elements have timestamp $t = 3$. Conversely, a *time-driven* $\mathcal{T}$ick would trigger a single evaluation.

The case of count windows is analogous, with the difference that the size and slide of windows are defined in terms of number of elements rather than time. SECRET identifies each element of the input stream with a unique *id* and defines a global order between the stream elements based on their *id*. In the case of count windows, the $\mathcal{S}$cope function is defined in terms of the parameter i_0—instead of t_0—which identifies the first *id* of the very first count window in the engine.[2]

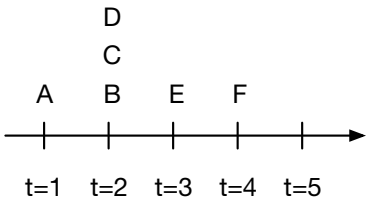
Notice that a corner case can occur in the case of *time-driven* $\mathcal{T}$ick associated to count windows. Indeed, multiple windows might close at the same point in time, leaving to the SPE the choice of which of them to report. Figure 3 exemplifies this situation for a count window with a size of two and a slide of two. Figure 3 denotes stream elements using upper case letters and orders them based on their event time. Element `A` has timestamp 1, and elements `B`, `C`, and `D` all have timestamp 2.

At time $t = 1$, no window closes and so the engine does not report. The same occurs at time $t = 2$. When the engine receives element `E` at time $t = 3$, all the elements for time $t < 3$ have been received—SECRET assumes that elements are processed in order. Thus, the system can process the elements `A`, `B`, `C`, and `D`, which fill two different windows of size two. Since the choice of the windows to report is engine specific, SECRET models this case by introducing a new $\mathcal{P}$ick function that encodes the selection.

As a final note, we observe that modern SPEs consider windows as special operators in their processing graph and can use multiple window operators in the same graph—see

[1] Since a window operator is defined in terms of a slide that is greater than zero, at any given point in time the active window for that operator is unique.

[2] In the remainder of the paper, we assume that the *id* associated to the very first element that enters the SPE after it starts has value 0. The *id* is increased by one for each incoming element.



t	report
1	{ }
2	{ }
3	?
4	{ }
5	{ E, F }

Fig. 3 Example of *time-driven* $\mathcal{T}$ick: count window with size 2 and slide 2

Fig. 1. On the other hand, SECRET focuses on windows and does not consider processing operators. Thus in this paper we model each window in isolation. We believe that this is not a severe limitation, since the overall semantics of a processing graph can be defined through the composition of its individual operators, and window operators are the most critical as they define the data slices upon which other operators get executed. At the same time, a complete modeling of the execution semantics of modern SPEs should also take into account the topology of the processing graph. We defer a detailed discussion of this and other possible limitations of the SECRET model to "Discussion".

Analysis of stream processing engines

This section models the execution semantics of five widely adopted modern SPEs using SECRET. We conduct an empirical analysis to determine the value of the parameters of SECRET—$\mathcal{S}$cope, $\mathcal{C}$ontent, $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{P}$ick—for each of the systems under analysis. The remainder of this section first presents the experimental methodology and then the systems under analysis and the results of our experiments for each of them.

Experimental methodology

We devise an experimental methodology to determine the parameters of the SECRET model for the systems under analysis. We build minimal processing graphs that include a single source and a single window operator w that outputs the entire content of w. We observe the time and content of each report from w.

Figure 4 presents the workflow defined in our methodology: the dashed block on the left of the figure represents the empirical experiments we perform to determine the functions $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{S}$cope, and $\mathcal{P}$ick. For each function, we list the concrete parameters that we need to identify, if any.

The behavior of $\mathcal{R}$eport depends on four independent flags: *content change*, *window close*, *non-empty content*, and *periodic*, which can be either active or inactive. $\mathcal{T}$ick can be either *tuple-driven* or *time-driven*. $\mathcal{S}$cope only depends on the parameter t_0—in the case of time windows—, or i_0—in the case of count windows. $\mathcal{P}$ick can be any function that takes a set s of windows and returns a subset of s.

To validate the correctness of our modeling, we rely on a simulator implemented by the authors of SECRET that we extend to capture the SPEs under analysis. For a given input and for given $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{S}$cope, and $\mathcal{P}$ick functions, the simulator generates the expected output. We compare the results of the simulator with the real results produced by each SPE using the same topologies and input data defined in the SECRET

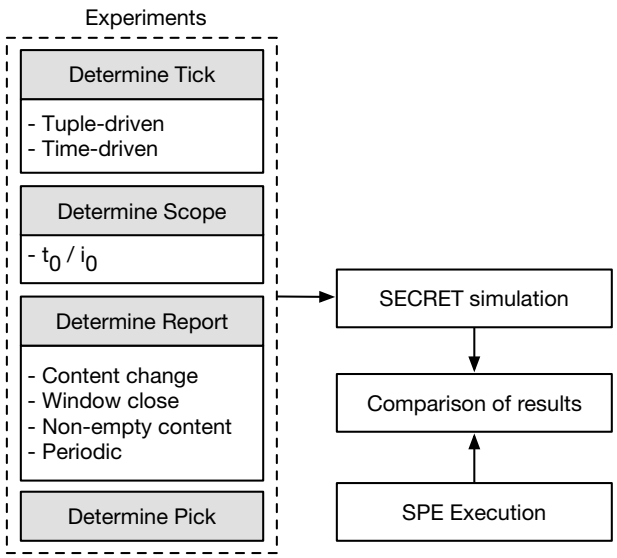


Fig. 4 Workflow of the experiments

paper [2], which exercise the corner cases of the modeled behavior. We made the code for the deployment and execution of all the experiments publicly available.[3] The entire codebase consists of about 3000 lines of code, mostly written in Java and including the original SECRET simulator. Plugging a new SPE only requires adapting the available examples to the API of that specific SPE. We hope this will promote future extensions to other SPEs.

We now proceed to describe the experiments we performed in detail. Unless otherwise specified in the description of the specific SPE, we deploy the SPE under analysis and the source of input data on the same machine. We configure the SPE as a stand alone component and we use network communication to interact with it. We submit input elements in timestamp order, always using FIFO channels to avoid reordering. SECRET does not capture problems related to clock skews and out of order elements that might emerge in distributed settings. Thus, these issues are outside the scope of the analysis and will be better discussed in "Discussion".

We adopt input elements in the form $\langle val, ts \rangle$, where val is a string value and ts is the timestamp of the element expressed in seconds. We design the experiments in such a way that timestamps well approximate the physical time measured on the wall clock time at the source. To do so, we pause the source between the submission of two consecutive elements $e_1 = \langle val_1, ts_1 \rangle$ and $e_2 = \langle val_2, ts_2 \rangle$ for $ts_2 - ts_1$ seconds. We keep the input rate low enough to guarantee that the SPE is never overloaded. Finally, we assume the variance of the latency between the source and the SPE to be negligible. Under these conditions, event time approximates well the processing time, thus making it possible to use SECRET also to analyze systems that adopt processing time semantics.

We discuss how we determine the $\mathcal{T}$ick, $\mathcal{R}$eport, and $\mathcal{S}$cope functions in the case of time windows, being the procedure for count windows analogous.

[3] https://github.com/deib-polimi/spes.

Determine $\mathcal{T}$ick

To determine the $\mathcal{T}$ick, we define a time window of size $\omega > 1$ and we submit more than ω elements all having the same timestamp. If the SPE reports more than once, then the $\mathcal{T}$ick is tuple-driven, meaning that the window advances even with elements that have the same timestamp, otherwise the $\mathcal{T}$ick is time-driven.

Determine $\mathcal{R}$eport

The semantics of report depends on four flags, and the SPE reports if (the constraints expressed in) *all* the flags are satisfied. We determine whether a flag *f* is satisfied by conducting experiments in which the results only depend on the satisfiability of *f*.

First, we build an experiment where *content change* and *non-empty content* do not influence the results. To do so, we submit elements with increasing timestamps, where the distance between the timestamp of two consecutive elements is a fixed value τ. We set a window of size τ and slide τ and we observe whether the system reports at every new tuple or only periodically, thus determining the value of the *periodic* flag. Next, we increase the size of the window to understand if the SPE reports only on *window close*. Second, we build an experiment in which the content of the window does not change across subsequent ticks, to determine the value of *content change*. As we discuss in the following, none of the systems under analysis exhibits periodic behaviors and none of them reports only on *content change*. Finally, we consider empty windows and analyze the reports of the SPE to determine the value of the *non-empty content* flag.

Determine $\mathcal{S}$cope

The $\mathcal{S}$cope function depends on the parameter t_0, which is the start time of the very first window considered by the SPE. Let us call ts_i the timestamp of the first element submitted to the SPE, ω the window size, and β the window slide. To understand the value of t_0, we submit elements to the SPE with a predefined frequency and we consider different combinations of ω, β, and ts_i. We use the results we obtain to infer the value of t_0 as a function of ω, β, and ts_i. In the case of count windows, we consider the parameter id_i instead of ts_i, which is the id of the very first element that enters the engine.

Determine $\mathcal{P}$ick

The $\mathcal{P}$ick function is only relevant in the case of count windows if the SPE under analysis provides time-based $\mathcal{T}$ick. As explained in "SECRET", this situation might lead to multiple windows closing simultaneously, and the $\mathcal{P}$ick function selects which of them to $\mathcal{R}$eport.

Most of the systems under analysis do not support count windows, and when count windows are available, they are always associated to a tuple-driven $\mathcal{T}$ick. Thus, the $\mathcal{P}$ick function is never used in the context of our analysis.

Flink

Flink[4] is an Apache project built on the Stratosphere research prototype [5] developed at the Technische Universität of Berlin. Flink has gained popularity in the last few years,

[4] https://flink.apache.org/.

becoming Apache Top Level Project in March 2014. It provides both a DataSet and a DataStream API to process static and streaming data, respectively. It also offers a number of libraries built on top of such APIs for disparate application domains, such as machine learning and graph processing.

We consider Flink 1.1.4 and we target the DataStream API. Flink processes streams of data using event time semantics. Time-related operators such as time-based windows use a *watermarking* mechanism [6] to reorder input elements before processing: an operator o sends a watermark with timestamp t to another operator o' to guarantee that o' will not receive from o any more items with timestamp lower or equal than t. When an operator receives a watermark with timestamp t from all its upstream operators, it can safely start processing all the elements up to time t.

Flink offers exactly once semantics even in the presence of node failures and implements fault tolerance using a distributed snapshot algorithm [7].

To model Flink in SECRET, we follow the methodology presented in "Experimental methodology", using a single source connected to a single window operator through a socket. Flink exposes some configuration parameters other than window size and slide to tune the behavior of time windows. For instance, developers can force Flink to implement a processing time semantics as opposed to the default event time semantics. In our experiments, we always adopt the default parameters and we derive the SECRET model accordingly. We defer to "Discussion" a detailed discussion of the implications of such parameters.

In Flink, each operator is responsible for providing the watermark to all its downstream operators. In particular, each source needs to submit a new watermark upon the delivery of a new element or periodically. In our experiments, we ingest input elements in event time order, so that the engine needs not wait for late, out of order elements. We make the source submit periodic watermarks with a frequency that is higher than the input rate. Each watermark is set to the timestamp of the last emitted element minus one second. This enables us to submit multiple elements with the same timestamp, for instance to determine the value of $\mathcal{T}$ick. All the elements with a timestamp τ are immediately processed when an element with timestamp greater than τ (and the corresponding watermark) is received, without waiting for out-of-order elements.

Table 1 lists the parameters of SECRET for Flink. In the case of time windows, the $\mathcal{T}$ick is time-driven and Flink reports on *window close* with *non-empty content*. The first window closes at time $ts_i + \beta - 1$, which means that it opens at time $t_0 = ts_i + \beta - \omega - 1$. In the case of count windows, Flink changes its $\mathcal{T}$ick to tuple-driven. As a consequence, it never experiences windows closing simultaneously and so it does not need to implement the $\mathcal{P}$ick function. The value of $i_0 = id_0 + \beta - \omega - 1$ is analogous to the t_0 extracted for time windows. Under the assumption that $t_0 = 0$ and $i_0 = 0$ the formulas for t_0 and i_0 can be simplified to $t_0 = i_0 = \beta - \omega - 1$.[5]

As a side note, we report that the behavior of Flink might differ from the behavior of the SECRET model when considering the very last active window before the system shuts down. Indeed, Flink flushes the active window when the system terminates, but this behavior is not captured by SECRET, which models the SPE assuming a steady state.

[5] SECRET assumes the domain of time and ids to be discrete. Timestamps and ids are allowed to be negative.

Table 1 Parameters identified for the SPEs under analysis

	$\mathcal{T}$ick	$\mathcal{R}$eport				$\mathcal{S}$cope	$\mathcal{P}$ick
		CC	WC	NE	P		
Time windows							
Flink	Time		✓	✓		$t_0 = ts_i + \beta - \omega - 1$	
Storm	Time		✓	✓		$t_0 = ts_i + \beta - \omega$	
Spark	Time		✓			$t_0 = \beta - \omega - 1$	
DataFlow	Time		✓	✓		$t_0 = ts_i + \beta - \omega - 1$	
Azure S.A.	Time		✓	✓		?	
Count windows							
Flink	Tuple		✓	✓		$i_0 = id_i + \beta - \omega - 1$	–
Storm	Tuple		✓	✓		$i_0 = id_i + \beta - \omega - 1$	–
Spark 1.6	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
DataFlow	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Azure S.A.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.

Storm

Storm[6] is a framework for distributed real-time computation acquired by Twitter in 2011 and released as an open source Apache project. In Storm, the developers specify the behavior of individual operators using imperative code and connect operators with each other to form a direct acyclic graph. Starting from version 1.0, Storm also offers a native support for time windows, which are the subject of our model. Each Storm operator processes stream elements one at a time and implements fault tolerance using a per-element acknowledgement and resubmission. This ensures that each operator processes each input element *at least once*, but does not guarantee *exactly once* processing. This aspect might affect the semantics of processing in the case of faults, but it is not modeled in SECRET. Thus, in the case of failures, the results produced by Storm might differ from those produced by its SECRET model.

We performed our experiments using Storm version 1.0.2. We implemented a topology with a single source and a single window operator connected through a TCP socket. Since Storm supports event time semantics and handles out of order arrival of elements through watermarking, we adopted the same strategy for the submission of watermarks as in Flink.

Table 1 lists the parameters to model Storm in SECRET. In the case of time windows, Storm presents the same $\mathcal{T}$ick and $\mathcal{R}$eport as Flink: it has a time-based $\mathcal{T}$ick and reports on *window close* with *non-empty content*. Storm's t_0 differs from Flink's t_0 by a fixed offset of one: while this difference might appear marginal, this means that the windowing behavior of Storm and Flink are different and thus the two engines produce different results when fed with identical input.

Initially, we started our experiments with Storm version 1.0.1. Using this version, we could not model count windows, since in the case of elements with identical timestamps Storm reported the last ω elements in the active window $\lfloor n/\omega \rfloor$ times, where ω is the size

[6] http://storm.apache.org.

of the window and n is the number of elements not-yet reported. We reported this suspicious behavior to the developers that recognized it as a bug. Storm version 1.0.2 integrates a bug fix[7] and we complete Table 1 accordingly. The observed behavior is identical to that of Flink. The detection of this bug highlights the importance of a formal model to analyze the execution semantics of SPEs. Thanks to the adoption of SECRET we could identify the presence of a misbehavior.

Spark

Spark [8] is a general-purpose cluster computing system. Initially developed at Berkeley and currently an Apache project, Spark is widely adopted in batch and stream processing applications that involve large volumes of data, also thanks to the availability of several application-specific libraries. Spark stores data in resilient distributed datasets (RDDs), a read-only, lazily evaluated, partitioned collections of records. RDDs are persisted on a distributed file system for fault tolerance and can be cached in main memory to enable low-latency computations. Spark keeps track of how RDDs are computed. If an RDD is lost due to some failure, its value is recomputed through the RDDs it depends on. This guarantees exactly once semantics even in the presence of node failures.

Spark models streaming computations as a series of stateless, deterministic batch computations on small batches of data—μ-batch. In particular, streams are discretized into a sequence of immutable, partitioned RDDs, which enable Spark to reuse the strategies and algorithms for batch computation also in streaming scenarios. This approach trades latency for throughput, since it delays a computation until a μ-batch is available. The μ-batch approach has also some consequences on the semantics of processing. Specifically, Spark only supports time windows having a size and slide that are multiple of the μ-batch size. This technological constraint is outside the scope of SECRET and thus is not captured in our model.

More significantly, Spark only supports processing time semantics and not event time semantics. Each operator considers the wall clock time of the machine it is running on and thus the order and time distance of the elements in a stream also depend on the processing capabilities of that specific machine. This means that the same input might produce different results in different deployments of the same processing topology. As discussed in "Experimental methodology", we designed the source such that its submission time is a good approximation of the value of its timestamp. This enables us to use the SECRET model, which relies on event time, also in the case of processing time. However, it is worth mentioning that this approach only works if the system is not overloaded: if the system instead introduces some processing delay, event time and processing time might differ, making our modeling imprecise.

In our experiments, we adopt Spark version 2.1. We configure Spark to use the minimum size of μ-batch of 1 ms and we connect a single source to the engine using a TCP socket. Spark only supports time windows and does not offer count windows. As shown in Table 1, Spark implements a time-based $\mathcal{T}$ick that reports on *window change* independently from the content of the window. The value of t_0 is different than in the other

[7] https://github.com/apache/storm/pull/1568.

systems we analyze. Indeed, Spark closes the first window at time $\beta - 1$ after the SPE starts up at time τ and also reports empty windows (thus the *non empty window* flag is not set).

SECRET cannot model the start up time τ of the system, since it can only reason about the event time as encoded in the timestamps of incoming elements, while τ can only be captured by considering the wall clock time of the machine in which the SPE runs. To solve this issue, in our experiments we assume that the SPE starts up at time 0, thus obtaining the value $t_0 = \beta - \omega - 1$. This highlights a limitation of SECRET that we better discuss in "Discussion".

Google Cloud Dataflow

Google offers the Google Cloud Dataflow[8] SPE as a service in the Google Cloud Platform. Google open-sourced the Software Development Kit for Dataflow, but not the underlying engine that remains proprietary.

Since Dataflow is only offered as a Cloud service, we had to adapt the experimental methodology. We communicate with the engine using the Pub/Sub messaging service.[9] Specifically, we deploy a topology composed of four operators: (1) a Pub/Sub *subscriber* that reads elements from an input topic; (2) a window operator; (3) a *reducer* that concatenates the content of each window into an output string; (4) the Pub/Sub *publisher* that writes the results of the reducer on an output topic. We submit elements by publishing them on the input topic and we read the results from the output topic.

By default, Dataflow applies event time semantics and adopts a watermarking mechanism to handle out of order. Watermarks are emitted by the very first operator that receives elements from outside the SPE (the *subscriber* in our topology). During the experiments, we observed a dependency between the input and the output rates. In particular, a low input rate could lead to unbounded output latency. We believe that a dynamic management of watermarks determines this behavior: in presence of a low input rate, the *subscriber* assumes a high latency in the arrival of elements, and thus becomes more conservative in the values of watermarks. As a consequence the values of watermarks do not advance and the engine stops processing new elements and emitting new output. We overcome this problem by attaching a long tail of additional elements at the end of each experiment, which ensures that we eventually receive all the output of that experiment.

Table 1 shows the SECRET parameters for Dataflow. Dataflow only supports time windows and does not provide count windows. The $\mathcal{R}$eport is *not-empty content* and on *window close*, while its $\mathcal{T}$ick is time-based. The $\mathcal{S}$cope is the same as in Flink $t_0 = ts_i + \beta - \omega - 1$.

Azure Stream analytics

Microsoft offers the Azure Stream snalytics SPE[10] as a Cloud service. The communication with the SPE can be realized using either a publish-subscribe service or a message

[8] https://cloud.google.com/dataflow/.

[9] https://cloud.google.com/pubsub/ in the Google Cloud Platform, which offers multicast communication on distinct channels called *topics*.

[10] https://azure.microsoft.com/en-us/services/stream-analytics/.

queuing service. In our experiments we submit elements through publish-subscribe primitives and we read output elements from a queue.

```
SELECT     System.Timestamp AS ts, Collect()
INTO       output-queue
FROM       input-hub TIMESTAMP BY tapp
GROUP BY   HoppingWindow(second, 4, 2)
```

Listing 1 An example of Stream analytics query

In Azure Stream analytics, the developer defines the processing tasks in the Stream Analytics Query Language, a SQL-like declarative language that is compiled to a graph of operators and deployed on the Cloud. Listing 1 shows the query we use in our experiments. The `GROUP BY` clause introduces a `HoppingWindow`, which is the Azure Stream Analytics implementation of a time based sliding window. The query extracts the timestamp of each element—stored in the `tapp` field—and returns the entire content of the window—`Collect()`—and the (event time) timestamp of the end of the window.

Azure Stream analytics supports event time semantics, where timestamps represent time in UTC. Developers can set a maximum time skew for late arrivals. When an element e enters the engine, the engine compares the ingestion time of e with the timestamp of e. If the difference between the two times is larger than the maximum time skew, then the engine drops e. Since the maximum time skew cannot be larger than few days, we could not use the same timestamps adopted for the other SPEs that start from 0. Instead, we shifted all the timestamp by δ, where δ is the UTC time when the experiment started, as extracted from the wall clock time of our local machine. This solution enables us to extract the parameters for the Azure Stream Analytics for $\mathcal{T}$ick and $\mathcal{R}$eport. We were not able to extract the value of t_0 since it probably depends on the ingestion time of the first element, which we cannot access from the Cloud platform.

As Table 1 shows, Azure Stream Analytics only supports time based windows, reports on *window close* with *non-empty content*, and presents a time-based $\mathcal{T}$ick.

Discussion

This section presents the key conclusions we draw from the modeling effort reported in "Analysis of stream processing engines". Under some assumptions that we better discuss in the remainder of this section, SECRET captures the semantics of windows in the SPEs we analyzed and highlights a general agreement on the $\mathcal{T}$ick and $\mathcal{R}$eport of windows, based on the following rules: (1) in the case of time windows, the $\mathcal{T}$ick is always time-driven; (2) in the case of count windows (if available), the $\mathcal{T}$ick is always tuple-driven; (3) all systems $\mathcal{R}$eport on *window close* and *non-empty content*, with the only exception of Spark that also reports empty windows.

Conversely, the systems under analysis present differences in terms of $\mathcal{S}$cope. In most systems the first active window closes after a slide $\beta - 1$ from the arrival of the first element, meaning that $t_0 = ts_i + \beta - \omega - 1$ ($i_o = id_i + \beta - \omega - 1$ in the case of count windows). Storm considers $t_0 = ts_i + \beta - \omega$, which is probably due to a different definition of windows that includes the open time and excludes the end time. Finally, Spark adopts a processing time semantics, and the position of windows is not related to the timestamp of the first element, but rather to start up time of the SPE.

While the different $\mathcal{R}$eport and t_0 in Spark and the different t_0 in Storm might appear small variations with respect to the behavior of the other SPEs, they lead to different execution semantics and thus to different results. This motivates the need for a formal model to capture the execution semantics of modern SPEs and highlight their differences.

Our analysis also sheds light on some aspects of modern distributed SPEs that SECRET cannot fully model. They are of great interest since (1) they hint at the key differences between "old generation" SPEs (those that SECRET was designed to model) and "modern" SPEs for distributed processing in cluster environments; (2) they suggest promising lines of investigation to build a more comprehensive model that captures all the relevant aspects of modern SPEs. We describe them in detail in the next sections.

Time model

SECRET assumes event time semantics and further assumes that all the stream elements enter the engine in order with respect to their timestamps. These assumptions were motivated by the nature of the stream processing systems available when SECRET was conceived, which were mainly centralized and specifically designed for continuous query answering. Stream elements often encoded occurrences of noteworthy facts in the application environments and queries typically included time boundaries. Because of this, windows—and time windows in particular—were perhaps the most relevant operators for these systems.

Conversely, modern SPEs such as the ones we consider in this paper are designed to perform generic computations on large volumes of streaming data. They do not consider time as a first class citizen and do not necessarily associate a timestamp to each and every stream element. In most cases, windowing constructs are not core building blocks of the engine, but rather operators developed on top of the base engine services to better support some application scenarios. This is for example the case of Storm, which offers windows only starting from recent versions.

Moving from these premises, it is not surprising that some systems, such as Spark, use processing time as their default time semantics. Under processing time semantics stream elements are processed by an operator in the order in which they enter that operator. The engine does not assume elements to be timestamped and thus it cannot exploit timestamps to learn the semantics of time of the application at hand.

The execution semantics of a single windowing operator under processing time can be approximated in event time if the following conditions hold: (1) the order in which the elements enter the SPE reflects the desired order for the application at hand. (2) The distance between the arrival of elements in processing time reflects their distance in event time; for instance, if the source that emits elements and the windowing operator are deployed on different physical machines, this means that the variance of the network latency between them is negligible. (3) The physical node that executes the window operator is not overloaded, and thus it does not introduce additional delay in the processing time.

The second and the third assumptions are the most critical, since they depend on runtime behaviors—processing speed, load, and network latency—that are not under the

control of the developers and difficult or impossible to achieve, as in the case of constant network latency.

As for the first assumption, it is reasonable to assume that individual external sources submit elements in some meaningful order, but without timestamps it is impossible to define a total order between elements produced at different external sources. If the developer wants to ensure event time semantics on top of a system that does not support it, she has to manually implement the application logic responsible for buffering and reordering input elements before processing.

Management of out-of-order elements

As discussed in the previous section, SECRET assumes input elements to enter the engine in order with respect to their associated timestamp. Instead, modern SPEs support out-of-order arrival of input elements, but differ in the way they manage such elements.

A common approach to deal with out of order consists in specifying the maximum delay for the arrival of elements and defer the processing until such a delay has elapsed. Yet, the semantics of processing in the case some element overcomes such maximum delay changes from system to system. For instance, as discussed in "Analysis of stream processing engines", Azure Stream analytics discards input elements that arrive too late with respect to the UTC wall clock time. These behaviors cannot be captured within the SECRET model.

Another approach consists in producing metadata at the sources to indicate when the produced elements can be safely processed without incurring the risk of late arrival of further out of order elements. The watermarks adopted in Flink are a concrete implementation of this more flexible approach [6] that enables each source to dynamically adjust the metadata it provides to the engine based on its current operating conditions. Also in this case, the semantics of processing in the case some elements violate the content of metadata might change from system to system.

Finally, other systems produce results in the form of mutable, time-annotated datasets. In the presence of out-of-order elements that alter the values of some results produced in the past, the engine retracts the previous output from the mutable dataset and substitutes it with the newly computed values. This is the case of the Kafka Stream system [9].

Graph of operators

SECRET considers the semantics of windows in isolation. This is motivated by the default processing model of the SPEs SECRET was designed to capture, which typically consists of a predefined, fixed structure with three steps [10]: (1) a stream-to-relation step that uses windows to select portions of the input stream; (2) a relation-to-relation step that performs the actual processing by only considering the content of windows; (3) a relation-to-stream step that converts back the results of the computation into stream elements. This fixed structure well serves the purpose of performing continuous queries on streaming data, but it is not flexible enough for general purpose computations.

Conversely, modern SPEs enable developers to build complex graphs of operators. How the structure of these graphs influences the execution semantics is outside the scope of SECRET and certainly relevant to fully model modern SPEs. For instance, some

SPEs admit cyclic topologies, but the way in which they implement cycles might differ. In general, the execution semantics might change depending on the way SPEs route elements between operators, since distributed deployments might affect the mutual order of stream elements as they move from operator to operator. Distribution becomes particularly relevant in the case of processing time semantics, where the presence of heterogeneous nodes, different processing speed, or different in latency between nodes might affect the overall behavior of the SPE in ways that cannot be predicted by only looking at the graph topology.

Finally, in modern SPEs windows are not special entities, but rather one of the possible operators that compose the processing graph. As a consequence, it becomes worth to investigate how different windows interact with each other based on their location in the graph.

Fault tolerance

Modern SPEs are designed to run on a multitude of physical nodes. In this setting, the probability of failure of at least one node is not negligible. Thus, SPEs include fault tolerance mechanisms to keep processing and producing results even in the presence of some failures.

Such mechanisms include per-element acknowledgments and retransmissions as in Storm, linage graph and recomputations as in Spark, and distributed snapshots as in Flink. Most significantly, different mechanisms provide different semantics and yield different results in the presence of failures. For instance, both Flink and Spark guarantee that each element is processed at each operator exactly once, which means that the results produced do not change in the case of failure. Conversely, Storm only guarantees at least once semantics, meaning that each operator can submit an element more than once to its downstream operators. Clearly, these differences affect the results produced by the engine in presence of faults and should be captured by a complete model of the system.

Even in the case of exactly once semantics, fault tolerance mechanisms affect the time when a result is produced. In the case of event time semantics, individual elements include a timestamp that defines the order and time distance between elements. In the case of processing time semantics, elements do not embed any notion of time and so their original order and relative time distance is lost in the case of faults. In other words, the assumptions we made to model systems with processing time semantics in SECRET no longer hold in the case of faults.

Summary and open challenges

The above discussion highlights the need to extend and complement SECRET to build a comprehensive model that fully captures the execution semantics of modern SPEs. Here we summarize the aspects of modern SPEs that demand for further modeling efforts and propose interesting directions for future research in this area.

First of all, SECRET assumes event time semantics. In the case of a single window in which all the elements are received in order, and in the presence of non overloaded machines, event time well approximates processing time.

However, the assumption of in order arrival of events might be unrealistic in several real world scenarios in which the clocks of different sources are not well synchronized or the channels between the sources and the SPE have different latency [3]. A proper analysis of this situation requires a model that takes into account the differences between the application time perceived at the sources and the time when elements are processed within the SPE.

Furthermore, under processing time semantics, the processing speed of the machines as well as temporary overloads might impact on the output produced. To capture these aspects, a model should take into account performance metrics. Given the intrinsic nondeterminism of performance measurements, we foresee the adoption of probabilistic models that encode the probability of the SPE to deviate from a default expected output.

Also, by solely relying on event time semantics, SECRET cannot model the aspects of an engine that depend on wall clock time. For instance, windows in Spark start to report after the engine starts up; similarly in Azure Stream Analytics, elements which time differs too much from the wall clock time of the physical machine get discarded. A comprehensive model that encodes both event time and wall clock time could also capture these behaviors.

Second, SECRET was designed to capture the semantics of individual windowing operators. This was motivated by the nature of the SPEs SECRET was designed for: those SPEs were mainly intended to answer continuous queries, where *a single* window was used to isolate the portion of the stream upon which the queries were evaluated. With well defined semantics for the queries, a precise modeling of the windowing behavior was then sufficient to capture the overall execution semantics of the engine.

Modern SPEs are designed to solve a larger class of problems, and offer programming abstractions that are suitable to encode general streaming computations. In most cases, the developer does not specify the processing task in terms of a high-level declarative query language, but rather explicitly defines the graph of operators that the input elements traverse to produce the output.

Windows still represent the most critical operators to model, since they accumulate the portions of the stream upon which other operators are executed. Nevertheless, a complete model that fully captures the execution semantics of modern SPEs needs to precisely encode the semantics of the operators graph formalism, defining its shape—for instance, whether loops are allowed—and behavior—for instance, how elements from multiple input streams are ordered within an operator.

Third, modern SPEs are designed to run on multiple nodes without a shared memory. In most cases the distribution of processing is transparent and does not impact on the output of the SPE. Yet in some cases the distribution might affect the execution semantics: for instance, we already discussed the consequences of heterogeneous processing capabilities under processing time semantics.

Thus, we believe that a precise model of SPEs should also be concerned with details about the processing infrastructure and the deployment of the operators on the physical nodes. For instance, it should consider whether the SPE enables the partitioning or the replication of some operators on multiple nodes to improve the performance, and how partitioning and replication might impact on the execution semantics of the engine.

Fourth, some modern SPEs introduce new types of windows that SECRET cannot model. For instance, Google Dataflow provides *session* windows, with boundaries not defined by the time or by the number of elements, but rather by the *content* of elements [3]. For instance, in a Web site monitoring scenario, a window might open upon receiving an element that indicates that a user started a particular operation and close when the operation ends.

Finally, modern SPEs introduce fault tolerance mechanisms to cope with the rather frequent hardware failure in the cluster platforms in which they operate. Different SPEs provide different guarantees in the presence of failures: in most cases, the SPE guarantees exactly once semantics, meaning that no loss or duplicate processing are possible and hence failure do not affect the execution semantics. However, some platforms offer weaker guarantees. For instance, Storm offers at least once semantics, in which duplicate processing of some elements is allowed. In this case, failures impact on the execution semantics and need to be modeled. This demands for a model that not only contemplates the presence of loss or duplicate elements in one operator, but also how loss and duplicate elements affect the entire processing graph.

Related work

We organize related work in three categories: "Processing streams of data" overviews the main approaches and systems to process streams of dynamic data, "Modeling stream processing" presents contributions that target the modeling of stream processing systems, and "Windowing approaches" discusses some proposals for windowing approaches that go beyond those captured by SECRET.

Processing streams of data

The last decade has seen an increasing interest in technologies to process streams of data, and several systems have been proposed both from the academia and from the industry. We distinguish two generations of SPEs. The first generation flourished in the mid 2000s and focuses on the definition of abstractions to query streams of data—as in data stream management systems (DSMSs)—or to detect situations of interest from streams of low-level information—as in complex event processing (CEP) systems. The interested reader can refer to the detailed survey of these systems by Cugola and Margara [11].

DSMSs usually rely on declarative query languages derived from SQL, which specify how incoming data have to be selected, aggregated, joined together, and modified, to produce one or more output streams [1]. The reference model of DSMSs has been defined in the seminal work on the continuous query language (CQL) [10]. In CQL, the processing of streams is split in three steps: first, *stream-to-relation* operators—windows—select a portion of each stream to implicitly create static database table. The actual computation takes place on these tables, using *relation-to-relation* (mostly SQL) operators. Finally, *relation-to-stream* operators generate new streams from tables, after data manipulation. Several variants and extensions have been proposed, but they all rely on the same general processing abstractions defined above. The declarative language of Azure Stream analytics that we analyzed in this paper also derives from this processing

model. The SECRET model we adopt in this paper was originally designed to capture the processing semantics of this kind of systems [2, 4].

The Aurora/Borealis DSMS first introduced the idea of defining the processing in terms of a directed graph of operators [12] and to deploy the operators on different physical nodes [13]. This approach deeply influenced the second generation of SPEs that we overview below and that are the subject of this paper.

CEP systems were developed in parallel to DSMSs and represent a different approach towards the analysis of streaming data, which targets the detection of situations of interest from patterns of primitive events [14, 15]. CEP systems typically consider the elements of a stream as notifications of event occurrences and express patterns using constraints on the content and time of occurrence of events [16, 17]. Interestingly, some CEP systems use interval timestamps. In this model, each data element is associated with two points in time that define the first and the last moment when it is valid [18, 19]. This model is never used in the modern SPEs that we consider in this paper and is not captured by SECRET.

More recently, the artificial intelligence and knowledge representation communities also started investigating streaming data posing an accent on integration and automated reasoning. This emerging field of research was named Stream Reasoning [20, 21]. In less then a decade, it has extended the Semantic Web stack with the RDF Stream data model, several continuous extensions to the SPARQL query language, and reasoning algorithms optimized for RDF streams [22, 23]. Systems in this area are often referred to as RDF Stream Processing (RSP) engines. The interested readers can refer to the working drafts of the W3C RSP Community Group.[11]

The second generation of SPEs has its roots in the research on Big Data and comprises systems designed to process large volumes of streaming data in cluster environments. The research on Big Data initially focused on static data and batch processing and proposed functional abstractions such as MapReduce [24] to automate the distribution of processing. Subsequent research increased the expressivity of MapReduce, enabling the developers to specify arbitrarily complex directed graphs of operators [8]. These systems assume long running computations and provide fault tolerance mechanisms to resume intermediate results if they are lost due to the failure of one or more machines in a large cluster [25].

The second generation of SPEs inherits the same processing model based on a graph of functional operators. Initially, the engines did not provide any built-in construct to express windows, and the developers had to implement windowing manually if required by the application at hand. This is for example the case of Storm prior to version 1.0 [26]. Given the complexity of time and window management, windows have become first class objects in all modern SPEs [3, 5, 27], and this motivates the need for a precise modeling and analysis of their behavior.

Modeling stream processing

Several formalisms have been proposed to specify the execution semantics of individual stream and event processing engines, ranging from automata [16, 28] to temporal logic [17], to event algebras and calculi [29, 30]. While these formalisms often capture

[11] http://www.w3.org/community/rsp/.

the semantics of time-based operators in general and window operators in particular, they are tailored to the specific features offered by the language they formalize and thus are not suitable to study the similarities and differences between heterogeneous stream processing systems.

Only few models have been proposed in the past to describe and compare some of the SPEs discussed above. The survey by Cugola and Margara introduces a framework of models that capture the key aspects of DSMSs and CEP systems [11], such as the data and processing models, the processing language and operators, and the semantics of time. Etzion and Niblett propose the event processing network (EPN) formalism to discuss CEP systems and event processing architectures [15]. An EPN is a directed graph where nodes represent operators and edges represent data flows between operators. These proposals, however, are descriptive and do not provide a formal ground to assess the commonalities and differences of the various SPEs. To the best of our knowledge, SECRET represents the first attempt to formally capture the time and window semantics of SPEs.

Interestingly, Dell'Aglio et al. define a model built on SECRET to capture the execution semantics of RSP engines [31]. The W3C RSP community group adopts it, together with LARS [32], as a reference model to define the semantics of RSP query language.

Given the relevance of performance—throughput and response time—for SPEs, several proposals aim to model performance characteristics of SPEs with the goal of predicting or improving some quality of service metrics or the allocation of resources [33–35]. These works are complementary to the proposal of this paper, since they focus on predicting the performance of SPEs rather than modeling their execution semantics.

Finally, some models have been proposed to describe at a higher level the organization of data processing architectures that include stream processing components. In this context, the seminal work by Nathan Marz proposes the well known *Lambda architecture* to meet the need for low latency, while guaranteeing exact and reliable results on "old" data [36]. The Lambda architecture was conceived when SPEs did not provide full support for distributed, fault-tolerant, and stateful computations and were used as a *speed layer* that could provide approximate results. The Lambda architecture couples this speed layer with a *batch layer* that runs periodic batch jobs to generate higher-latency, exact results. When the data is queried, the *serving layer* encapsulates the complex logic that decides whether to serve the results of the speed layer—recent, but possibly inaccurate—or those of the batch layer—accurate, but possibly outdated. More recent proposals are questioning this type of architecture because of its complexity and high maintenance costs, and foster the development of stream-only architectures, where the SPE plays a more central role.[12] All these models study the integration of SPEs within a data processing architecture, and do not consider the detailed execution semantics of the SPE, as SECRET does.

Windowing approaches

Some proposals target the definition of novel approaches and formalisms for windows to improve expressivity and flexibility. As mentioned in "Discussion", Google Dataflow and

[12] http://milinda.pathirage.org/kappa-architecture.com/.

Apache Flink introduced the concept of session windows [3], in which the boundaries of each window are defined based on the content of the input elements. Developers provide a function that associates a session identifier to each input elements, and windows group all the elements with the same session identifier. Li et al. [37] proposed a similar approach to define windows based on the content of input elements. They adopt this formalism to define an effective evaluation strategy for window aggregates. Frames [38] are content-defined windows that provide the developers with built in functions to simplify the statistical analysis of data.

Predicate windows [39] predicate on the content of an input element to determine whether it has to be considered as new information, or as an update (or deletion) of existing information. They were introduced to define views and to support view maintenance in DSMSs. Predicate windows are more flexible since they are not append-only, but they enable incoming elements to overwrite part of a window content. This approach is also at the base of some modern distributed stream processing systems, such as the Apache Kafka [9] message broker.

The advent of new types of windows will demand for novel models that can extend or complement SECRET.

Conclusions

This paper studies the execution semantics of modern distributed SPEs focusing on the key notions of time and windows. Our analysis grounds on the SECRET model that was developed in 2010 to capture the semantics of the SPEs available at that time. On the one hand, SECRET can capture the window behavior of most modern SPEs, which indicates that the same abstractions that were introduced in early systems are still adopted. The analysis highlights a general agreement on the semantics of time windows, supported by all the systems we analyzed, and count windows, present only in few systems. On the other hand, modern SPEs are far more complex that the systems SECRET was designed for, and a precise understanding of their behavior demands for additional modeling efforts to capture some key aspects that influence the processing semantics, such as the effect of distribution and out of order arrival of elements.

The paper lays the foundations for a precise understanding of modern distributed SPEs by investigating their time and window behavior, by shading light on the key aspects that cannot be captured with today's models, and by drawing a road map for future research efforts.

Authors' contributions
All the authors carried out the conception and the design of the research. LA and RT carried out the experimental evaluation. All the authors contributed to the interpretation of the data and to the writing of the paper. All authors read and approved the final manuscript.

Acknowledgements
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Availability of data and materials
The datasets supporting the conclusions of this article are available in the GitHub repository, in https://github.com/deib-polimi/spes.

Consent for publication
The authors declare that they consent the publication of the manuscript.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 10 January 2017 Accepted: 13 April 2017
Published online: 26 April 2017

References

1. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. Proceedings of the symposium on principles of database systems. PODS'02. New York: ACM; 2002. p. 1–16.
2. Botan I, Derakhshan R, Dindar N, Haas L, Miller RJ, Tatbul N. Secret: a model for analysis of the execution semantics of stream processing systems. VLDB J. 2010;3(1–2):232–43.
3. Akidau T, Bradshaw R, Chambers C, Chernyak S, Fernández-Moctezuma RJ, Lax R, McVeety S, Mills D, Perry F, Schmidt E, Whittle S. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. VLDB J. 2015;8(12):1792–803.
4. Dindar N, Tatbul N, Miller RJ, Haas LM, Botan I. Modeling the execution semantics of stream processing engines with secret. VLDB J. 2013;22(4):421–46.
5. Alexandrov A, Bergmann R, Ewen S, Freytag J-C, Hueske F, Heise A, Kao O, Leich M, Leser U, Markl V, Naumann F, Peters M, Rheinländer A, Sax MJ, Schelter S, Höger M, Tzoumas K, Warneke D. The stratosphere platform for big data analytics. VLDB J. 2014;23(6):939–64.
6. Akidau T, Balikov A, Bekiroğlu K, Chernyak S, Haberman J, Lax R, McVeety S, Mills D, Nordstrom P, Whittle S. Millwheel: fault-tolerant stream processing at internet scale. Proc VLDB. 2013;6(11):1033–44.
7. Schelter S, Ewen S, Tzoumas K, Markl V. "All roads lead to rome": optimistic recovery for distributed iterative data processing. Proceedings of the international conference on information and knowledge management, CIKM'13. New York: ACM; 2013. p. 1919–28.
8. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. Proceedings of the conference on hot topics in cloud computing. HotCloud'10. Berkeley: USENIX Association; 2010. p. 10.
9. Wang G, Koshy J, Subramanian S, Paramasivam K, Zadeh M, Narkhede N, Rao J, Kreps J, Stein J. Building a replicated logging system with Apache Kafka. Proc VLDB Endow. 2015;8(12):1654–5.
10. Arasu A, Babu S, Widom J. The cql continuous query language: semantic foundations and query execution. VLDB J. 2006;15(2):121–42.
11. Cugola G, Margara A. Processing flows of information: from data stream to complex event processing. ACM Comput Surv. 2012;44(3):15–11562.
12. Abadi DJ, 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. VLDB J. 2003;12(2):120–39.
13. Abadi DJ, Ahmad Y, Balazinska M, Cherniack M, Hwang J-h, Lindner W, Maskey AS, Rasin E, Ryvkina E, Tatbul N, Xing Y, Zdonik S. The design of the borealis stream processing engine. In: Proceedings of the conference on innovative data systems research. CIDR '05; 2005. p. 277–89.
14. Luckham DC. The power of events: an introduction to complex event processing in distributed enterprise systems. Boston: Addison-Wesley; 2001.
15. Etzion O, Niblett P. Event processing in action. Greenwich: Manning Publications; 2010.
16. Brenna L, Demers A, Gehrke J, Hong M, Ossher J, Panda B, Riedewald M, Thatte M, White W. Cayuga: a high-performance event processing engine. Proceedings of the international conference on management of data. SIGMOD'07. New York: ACM; 2007. p. 1100–2.
17. Cugola G, Margara A. Tesla: a formally defined event specification language. Proceedings of the international conference on distributed event-based systems. DEBS'10. New York: ACM; 2010. p. 50–61.
18. White W, Riedewald M, Gehrke J, Demers A. What is "next" in event processing? Proceedings of the symposium on principles of database systems. PODS'07. New York: ACM; 2007. p. 263–72.
19. Schultz-Møller NP, Migliavacca M, Pietzuch P. Distributed complex event processing with query rewriting. Proceedings of the international conference on distributed event-based systems. DEBS'09. New York: ACM; 2009. p. 4.
20. Valle Della E, Ceri S, van Harmelen F, Fensel D. It's a streaming world! Reasoning upon rapidly changing information. IEEE Intell Syst. 2009;24(6):83–9.
21. Margara A, Urbani J, van Harmelen F, Bal H. Streaming the web. J Web Semant. 2014;25(C):24–44.
22. Barbieri DF, Braga D, Ceri S, Della Valle E, Grossniklaus M. Querying rdf streams with c-sparql. Proc Int Conf Manag Data. 2010;39(1):20–6.
23. Anicic D, Fodor P, Rudolph S, Stojanovic N. EP-SPARQL: a unified language for event processing and stream reasoning. Proceedings of the international conference on world wide web. WWW'11. New York: ACM; 2011. p. 635–44.
24. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
25. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Proceedings of the conference on networked systems design and implementation. NSDI'12. Berkeley: USENIX Association; 2012. p. 2.
26. Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, Bhagat N, Mittal S, Ryaboy D. Storm@twitter. Proceedings of the international conference on management of data. SIGMOD'14. New York: ACM; 2014. p. 147–56.
27. Koliousis A, Weidlich M, Castro Fernandez R, Wolf AL, Costa P, Pietzuch P. Saber: window-based hybrid stream processing for heterogeneous architectures. Proceedings of the international conference on management of data. SIGMOD'16. New York: ACM; 2016. p. 555–69.
28. Agrawal J, Diao Y, Gyllstrom D, Immerman N. Efficient pattern matching over event streams. Proceedings of the international conference on management of data., SIGMOD'08. New York: ACM; 2008. p. 147–60.

29. Artikis A, Sergot M, Paliouras G. An event calculus for event recognition. IEEE Trans Knowl Data Eng. 2015;27(4):895–908.
30. Hinze A, Voisard A. Eva: an event algebra supporting complex event specification. Inf Syst. 2015;48:1–25.
31. Dell'Aglio D, Della Valle E, Calbimonte J-P, Corcho O. RSP-QL semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. Int J Semant Web Inf Syst. 2014;10(4):17–44.
32. Beck H, Dao-Tran M, Eiter T, Fink M. LARS: a logic-based framework for analyzing reasoning over streams. Proceedings of the AAAI conference on artificial intelligence. AAAI '15. Palo Alto: AAAI Press; 2015. p. 1431–8.
33. Cardellini V, Grassi V, Lo Presti F, Nardelli M. Optimal operator placement for distributed stream processing applications. Proceedings of the international conference on distributed and event-based systems., DEBS '16. New York: ACM; 2016. p. 69–80.
34. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M. Network-aware operator placement for stream-processing systems. In: Proceedings of the international conference on data engineering. ICDE '06. Washington, DC: IEEE; 2006. p. 49.
35. Basanta-Val P, Audsley NC, Wellings AJ, Gray I, Fernndez-Garca N. Architecting time-critical big-data systems. IEEE Trans Big Data. 2016;2(4):310–24.
36. Marz N, Warren J. Big Data: principles and best practices of scalable realtime data systems. Greenwich: Manning Publications Co.; 2015.
37. Li J, Maier D, Tufte K, Papadimos V, Tucker PA. Semantics and evaluation techniques for window aggregates in data streams. Proceedings of the international conference on management of data., SIGMOD '05. New York: ACM; 2005. p. 311–22.
38. Grossniklaus M, Maier D, Miller J, Moorthy S, Tufte K. Frames: data-driven windows. Proceedings of the international conference on distributed and event-based systems. DEBS '16. New York: ACM; 2016. p. 13–24.
39. Ghanem TM, Elmagarmid AK, Larson P-A, Aref WG. Supporting views in data stream management systems. ACM Trans Database Syst. 2008;35(1):1–1147.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com