A Model for Continuous Query Latencies in Data Streams

Roberto Baldoni, Giuseppe Antonio Di Luna, Donatella Firmani and Giorgia Lodi
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Universitá degli Studi di Roma "La Sapienza"
baldoni,firmani,lodi@dis.uniroma1.it, g.a.diluna@gmail.com

ABSTRACT

In this paper we propose a formal model for characterizing latencies affecting the computation of a continuous query either in a Data Stream Management System (DSMS) or in a Complex Event Processing (CEP) system. In the model, a query can be thought of as constructed out of basic Event Processing Units (EPUs) interconnected among themselves. EPUs are modeled considering just few parameters, used to define the EPU processing logic. In order to model the continuous query we use an acyclic directed (data-flow) graph whose nodes are the EPUs and edges represent the flow of information (events) processed by the EPUs themselves.
The outcome of this model is to associate with each data-flow graph a set of latency metrics, namely reactivity, activity, and output latencies, and a complexity measure - that we call data-flow graph complexity - representing the input dimension required to produce an output event.
The proposed model can be used to compare and contrast different data-flow graphs in order to assess their latency metrics. This is a crucial step in selecting one of such graphs that meets at best the latency requirements imposed by the programmer before its actual submission to a DSMS or to a CEP system. Furthermore, the model can be considered an effective mean through which formally comparing data-flow graphs and predicting their behavior before an actual experimental validation phase.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures, product metrics*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms

Algorithms, Theory, Performance, Measurement

1. INTRODUCTION

In recent years we have witnessed an increased adoption of Complex Event Processing (CEP) and Data Stream Management Systems (DSMS) in several heterogeneous application domains such as anomaly detection, environmental monitoring, and stock prices analysis, just to name a few. This is due to the increasing need of processing on-the-fly information generated by on-line information sources rather than use the typical database management system processing style that first stores and then processes the information.

In such systems a programmer defines a set of functions, i.e., the algorithm, which describes how incoming flows of information have to be processed to timely produce new flows as outputs (in the literature such data-flows are also referred to as data-streams). Therefore, given a specific problem (expressed in a form of the query) that takes several data streams as inputs, programmers could derive several set of functions and several distinct data-flow graphs that solve the problem equivalently, thus generating the same output.

The aim of this paper is to assess which could be the best data-flow graph according to some QoS requirements expressed by the programmer. The assessment should be done independently from the system in which it will be implemented and from the physical infrastructure in which it will be executed.

Our approach is to model the behavior of this set of functions, under the assumption that the processing time is shorter than the length in time of the input. This is motivated by several factors: (i) our analysis is independent of the specific computing platform; (ii) many streaming applications are built to work practically under this assumption (e.g., command and control application, collaborative security etc.) (iii) experiments show that existing complex event processing systems (e.g., Esper [1]) are able to cope with very high input event rate without showing significant performance degradation.

Specifically, we model the functions that solve the query through what we call a *query data-flow graph*, i.e., a directed graph G whose nodes are basic queries, i.e., EPUs, incoming edges to a node represent input streams and outcoming edges are output streams from a node.

Over the data-flow graph we define several latency metrics, namely the output latency, reactivity latency and activity latency as well as a complexity measure that defines the input dimension necessary to produce an event as output. As depicted in Figure 1, the activity latency takes into account the distance in time between the first event in input to G and the last event output by G. The output latency is the distance in time between the first event input to G and the first event output by G while the reactivity latency takes into account the distance in time between the last event input to G that is necessary to produce an output and the first event output by G. The different latencies represent different QoS aspects of G that a programmer could be interested in exploiting when specifying the processing.

We provide a formal model to evaluate such latencies and the input complexity on G. The model has a noteworthy practical impact: it could be an effective instrument in order to compare different query graphs designed by a programmer solving the same problem; furthermore, this comparison process can be automatized, i.e., it can be implemented in a software tool that, given a query specified in some high-level language and some specific latency requirements (e.g., minimum reactivity latency) provided by the programmer, it is able to firstly produce several query graphs solving the same high-level query and then rank such graphs according to user requirements. The selected data-flow graph could be submitted to either a specific Data Stream Management System (DSMS) (e.g., [3]) or a CEP system (e.g., [2, 1]).

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the system model we propose. Section 4 presents the four metrics of interest obtained using the model. Section 5 discusses the analysis of the four metrics and an example of applicability of the model, finally, Section 6 concludes the paper.

2. RELATED WORK

All Data Stream Management Systems (DSMSs) include several techniques to allow each stream query to meet its QoS requirements such as tuple latency, memory usage, throughput. Such requirements can be specified with each query submitted to the DSMS. They generate conflicts inside the DSMS that are handled by specific components (e.g., scheduler, run-time optimizer, load shedding etc.) that cooperate to optimize and balance each metric of each query so that anyone can meet target QoS requirements. Noteworthy examples of such systems are AURORA [3] and STREAM [4]. Our model associates different latency metrics with a single query graph based on its structure, only. In doing so, we can select the best query graph solving a problem with respect to specific latency metrics and then submit it to the DSMS. Our model then complements at design time the work done by a DSMS to optimize QoS metrics.

The idea of using data-flow graph to model a continuous streaming computation has been explored by several systems (e.g., AURORA [3], System S [2], Esper [1]). As an example, in AURORA a continuous query is represented by a directed acyclic graph whose nodes are boxes. Each box represents a processing operation and directed edges represent input tuple streams to box (resp. output tuple stream from a box). Boxes are combined and reordered by the scheduler during a query optimization phase in order to obtain the best performances.

In [9] and [10] the authors analyze the problem of computing performance metrics of relational operators in a streaming query and provide optimization frameworks for query evaluation planners. The problem of avoiding overload of operators in distributed environments and selecting a resilient operator placement plan, is studied in [11] and [12], whereas a particular cost estimation technique applicable to both the above mentioned problems, is studied in [6].

Our approach is to propose a cost estimation technique of an algorithm solving a problem expressed in terms of continuous query computation, that is independent from the effective implementation on a DSMS. The algorithm, after its design, optimized by the model discussed in this paper, can be implemented in a DSMS and then (i) it may be converted in a data-flow plan of operators; (ii) the operators may themselves be distributed amongst the available nodes (machines) in different ways.

As a matter of fact, the previously mentioned works appear to be very useful when modeling the plan selection and the distributed operator placement inside a DSMS, in order to access streaming data efficiently and to balance load and manage resources of operators effectively.

3. SYSTEM MODEL

Event Processing Unit.

An Event Processing Unit (EPU) is a function that takes streams as input, performs a computation and originates a *single* stream as output for downstream consumption. An EPU can be an algebra operator (e.g., Aurora [3]) or a relational operator (e.g., [5]) or any user-defined operator as in [8].

If the EPU consumes more than a stream to produce the output stream, depending on the way it process the different streams, it may assume two different behaviors:

- *All-Streams Batch Processing* (ASB): the ASB EPU computes a function that needs the existence of events on *all* input streams to be computed; for instance, the function can be the logical `and`.
- *All-Streams Online Processing* (ASO): the ASO EPU computes a function that needs the existence of events on *at least one* input streams to be computed; for instance, the function can be the logical `or`.

If the EPU consumes at least a stream to produce the output stream, depending on the domain of the function it computes, it may assume two different behaviors, independently of the previous categorization:

- *Event Based* (EB): the EB EPU computes a function defined - for each input stream - on an n-dimensional domain, where n is the number of consumed events; for instance, the function can be the algebraic `sum` on n-dimensional vectors.
- *Time Based* (TB): the TB EPU computes a function defined on an temporal domain; the function is computed over a set of events happened in a given time interval; as an example, the function can be the relational `count` on the last 2 seconds.

An EPU is called *producer* when it does not consume any input stream in order to produce the output stream. In contrast, an EPU is called *consumer* if there are no other EPUs consuming its output stream.

Data-Flow Graph.

Given a query expressed in natural language, where inputs are streams of events, the query can be solved by defining a set of EPUs and their connection relationships.

To this end, we define *data-flow graph* a directed acyclic graph $G = (V, E)$ where V contains all the EPU *nodes* and

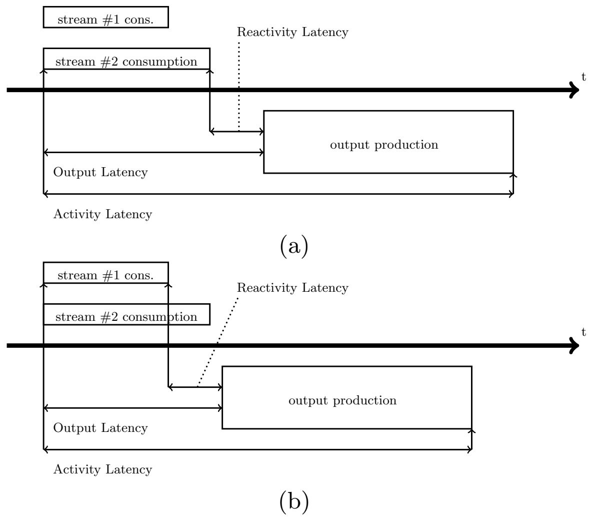


Figure 1: Output Latency, Activity Latency and Reactivity Latency: (a) ASB processing; (b) ASO processing. In relation to the time/event based definition of an EPU, the consumed sets of events can depend on either the time or the number of the events that they contain.

Process a raw market data feed: report throughput statistics and detect when the data rate of a feed falls off unexpectedly. A rate fall-off may mean that the data is stale and we want to alert when there is a possible problem with the feed.

Figure 2: An example of query expressed in natural language, `MarketDataFeedMonitor` [1].

in E there exists an *edge* (v, u) if and only if there exists an EPU $v \in V$ that produces an event stream which is in turn consumed by an EPU $u \in V$.

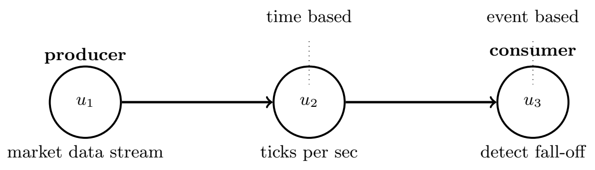
We define $I(u) = \{v_1, v_2 \dots\}$ as the set of head endpoints adjacent to node u; hence, $|I(u)|$ is equal to the indegree of u. Each EPU is modeled as a *node* with $w_u = |I(u)|$ incoming edges and w_o outcoming edges, representing the w_u EPUs that produce the event streams consumed by u, and the w_o EPUs that consume the event stream produced by u.

Fig. 3 shows an example of data flow graph for the query described in Fig. 2.

EPU Parameters.

Each EPU node is characterized by a set of parameters that describe how u transforms the event streams and that represent the domain on which the metrics represented in Fig. 1 are defined.

Given the EPU u, the parameters are the following.
The *constant* $t_u(v)$, where $v \in I(u)$, is defined as the time interval length in which u, **if it is time-based**, consumes events in order to produce a *single* event. In particular we define two *constants* $\hat{t}_u(v)$ and $\check{t}_u(v)$ such that $t_u(v) \in [\hat{t}_u(v), \check{t}_u(v)]$.
The *constant* $n(u)$ is defined as the dimension of codomain of the function computed by the EPU, i.e., the number of events produced by the EPU by evaluating the function.



(a)

EPU	operation

u_1	`String symbol;`
	`FeedEnum feed;`
	`double bidPrice;`
	`double askPrice;`

u_2	`insert into TicksPerSecond`
	`select feed, count(*) as cnt`
	`from MarketDataEvent.win:time_batch(1 second)`
	`group by feed`

u_3	`select feed, avg(cnt) as avgCnt, cnt as feedCnt`
	`from TicksPerSecond.win:time(10 seconds)`
	`group by feed`
	`having cnt < avg(cnt) * 0.75`

(b)

Figure 3: An example of solution for the query `MarketDataFeedMonitor` (Fig. 2): (a) data-flow graph; (b) description of the three EPUs. The functions computed by the EPUs are described using the Event Processing Language (EPL) [1].

The *variable* $n_u(v)$, where $v \in I(u)$, is defined as the *average* number of events that u, **if it is event-based**, consumes in order to produce a *single* event. In particular, we define two *constants*[1] $\hat{n}_u(v)$ and $\check{n}_u(v)$ such that $n_u(v) \in [\hat{n}_u(v), \check{n}_u(v)]$, i.e., that represent the minimum and the maximum value that $n_u(v)$ can have. Depending on the value of the constant $\hat{n}_u(v)$, u may assume two different behaviors, independently of the previously discussed categorizations:

- *Per-Stream Batch Processing* (PSB): the PSB EPU cannot produce an event without waiting for the consumption of more than one event, i.e.:

$$\hat{n}_u(v) > 1 \texttt{ or } \frac{1}{n(u)} < \hat{n}_u(v) < 1 \tag{1}$$

- *Per-Stream Online Processing* (PSO): the PSO EPU can produce an event after having consumed even a single event, i.e.:

$$\hat{n}_u(v) = 1 \texttt{ or } \hat{n}_u(v) < \frac{1}{n(u)} \tag{2}$$

The *variable* $p(u)$ is defined as the *average* time required by the EPU to evaluate the function.

4. EPU METRICS

Evaluating EPU metrics.

In order to evaluate the latency metrics we need to compute for the EPU u, and for each EPU $v \in I(u)$, some basic quantities that describe the elementary aspects of how the EPU transforms the input streams in the output stream. These quantities are the input rate $\rho_u(v)$, output rate $\rho(u)$,

[1]The value of the constants is supposed to be an input of our model.

input silence period $\sigma_u(v)$ and output silence period $\sigma(u)$, each described in the following.
If the EPU u is **time-based**, the *input rate* $\rho_u(v)$ is defined with respect to each input channel and it represents the average frequency at which the input set is produced by v.

$$\rho_u(v) \quad = \quad \frac{n_u(v)}{\frac{n_u(v)}{CHR} + \sigma(v)(\frac{n_u(v)}{n(v)} - 1)} \tag{3}$$

CHR is a constant representing the maximum rate allowed by the channel bandwidth connecting an EPU v to an EPU u, and $(\frac{n_u(v)}{n(v)} - 1)$ is the number of *function evaluations* that v must perform in order to have a single function evaluation of u.

With the term *function evaluation* of an EPU, we refer to a single execution of the process that the EPU iteratively runs in order to continuously process the input stream. Indeed a function evaluation of an EPU u corresponds to the production of $n(v)$ events (see definition of EPU parameters, in Sec. 3).
The *output rate* $\rho(u)$ is defined with respect to the output channel, and it represents the average frequency at which the output set is produced by v.

$$\rho(u) \quad = \quad \frac{n(u)}{CHR} \tag{4}$$

Once the function is correctly evaluated, by the EPU u, in a time period that is represented by $p(u)$, it is assumed that each of the $n(u)$ output events is produced in $\frac{1}{CHR}$ time units and that the $n(u)$ output events are produced sequentially without any interruption.
The *input silence period* $\sigma_u(v)$ is defined with respect to each input channel, and it represents the time interval between two input sets produced by v. The two input sets correspond to two consecutive evaluations of the function computed by u.

$$\sigma_u(v) \quad = \quad \begin{cases} \sigma(v) & \text{if } u \text{ is EB} \wedge n_u(v) \texttt{ mod } n(v) = 0 \\ p(v) & \text{if } u \text{ is EB} \wedge n_u(v) \texttt{ mod } n(v) > 0 \\ 0 & \text{otherwise, i.e., } u \text{ is TB} \end{cases} \tag{5}$$

The *output silence period* $\sigma(u)$ is defined with respect to the output channel, and it represents the time interval between two output sets produced by u. The two output sets correspond to two evaluations of the function computed by u, as illustrated in Fig. 4.

$$\sigma(u) \quad = \quad \begin{cases} OL(u) + \sigma_u(\tilde{v}) - (AL(u) - \frac{n_u(\tilde{v})}{\rho_u(\tilde{v})}) & \text{if } u \text{ is EB} \\ OL(u) & \text{otherwise} \end{cases} \tag{6}$$

Where $\tilde{v} \in I(u)$ is an EPU such that:

$$\tilde{v} = \underset{v}{\text{argmin}} \frac{n_u(v)}{\rho_u(v)} + \sigma_u(v) \tag{7}$$

Moreover, $OL(u)$ is the output latency of the EPU and $AL(u)$ is its activity latency, as discussed below.

Output Latency.

The *output latency* of an EPU, denoted with $OL(u)$, is defined as the time elapsed between the **beginning** of the input event streams production and the **beginning** of the output event stream production. More in detail, if u is event based the beginning time is the production of the first input

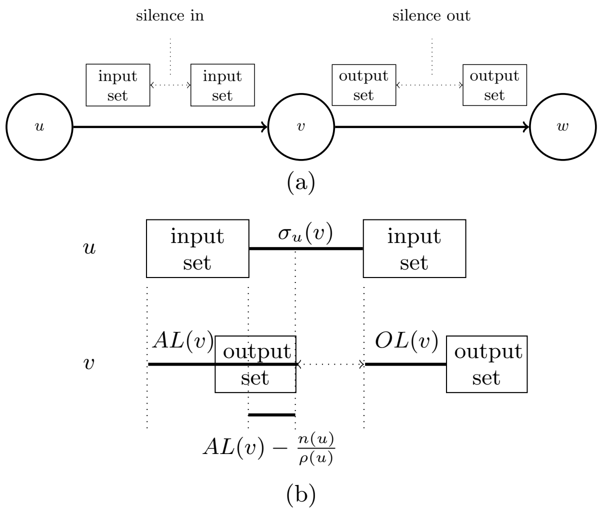


Figure 4: An example of evaluation of EPU v's output silence period starting from the input silence period of an EPU u.

event; in contrast, if u is time based, the beginning time is the starting time of the time interval. Formally:

$$\tau_u(v) \quad = \quad \begin{cases} \frac{n_u(v)}{\rho_u(v)} \cdot n(u) & \text{if } u \text{ EB} \\ t_u(v) & \text{otherwise} \end{cases} \tag{8}$$

$$OL(u) \quad = \quad \begin{cases} \max_{v \in I(u)} \tau_u(v) + p(u) & \text{if } u \text{ ASB} \\ \min_{v \in I(u)} \tau_u(v) + p(u) & \text{otherwise} \end{cases} \tag{9}$$

The value of $OL(u)$ when $I(u) = \emptyset$ is $p(u)$ by convention.

Note that in the $OL(u)$ definition we do not consider that the beginning time may differ with respect to the $|I(u)|$ different input streams. This is due to the fact that in our model this quantity represents the analysis of the entire data-flow graph and not that of the single EPU. As discussed in Sec. 5, the role that defines how to sum the EPU output latencies through the data-flow graph takes this time interval implicitly into account.

Activity Latency.

The *activity latency* of an EPU, denoted with $AL(u)$, is defined as the time elapsed between the **beginning** of the input event streams production and the **end** of the output event stream production. Therefore the activity latency is equal to the output latency to which adding the time taken to produce the output event stream. Formally:

$$AL(u) = OL(u) + \frac{n(u)}{\rho(u)} \tag{10}$$

Reactivity Latency.

The *reactivity latency* of an EPU, denoted with $RL(u)$, is defined as the time elapsed between the **end** of the input event streams production and the **beginning** of the output event stream production. It represents the time required to trigger the production of the output event stream under the assumption that the last event necessary to produce it has

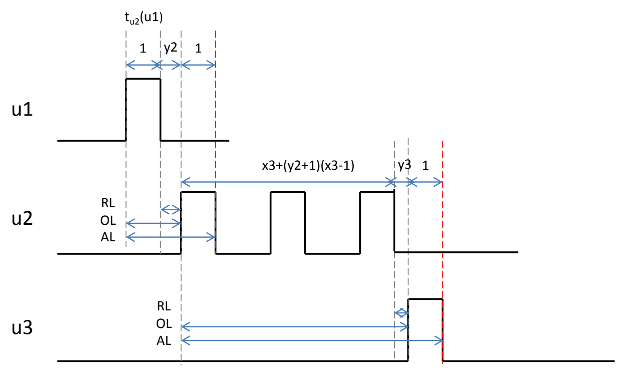


Figure 5: Timing diagram of the EPUs of the data-flow graph illustrated in Fig. 3 for the query `MarketDataFeedMonitor` (Fig. 2).

just been consumed. Formally:

$$RL(u) = \begin{cases} OL(u) - \max_{v \in I(u)} \tau_u(v) & \text{if } u \text{ ASB} \\ OL(u) - \min_{v \in I(u)} \tau_u(v) & \text{otherwise} \end{cases} \quad (11)$$

$$= p(u) \quad (12)$$

Note that, by definition, the reactivity latency of an EPU u is always equal to $p(u)$, whereas the definition of reactivity latency for a data-flow graph is something more complex than a sum of $p(*)$ terms, as discussed in Section 5. The definition of $\tau_u(v)$ is the same as in the definition of $OL(u)$.

EPU Complexity.

The *complexity* of an EPU u, denoted with $C(u)$, is defined as the *average* number of events that u, **if it is event-based**, consumes in order to produce the $n(u)$ events of its output set, with respect to the EPU $v \in I(u)$ that represents the worst case, i.e., that maximizes this quantity. Formally:

$$C(u) = \begin{cases} \max_{v \in I(u)} n_u(v) \cdot n(u) & \text{if } u \text{ EB} \\ 0 & \text{otherwise} \end{cases} \quad (13)$$

The value of $C(u)$ when $I(u) = \emptyset$ is 1 by convention[2].

Fig. 5 and Tab. 1 show, respectively, the timing properties and the evaluation of the metrics of the EPUs of the data-flow graph of Fig. 3, for the query `MarketDataFeedMonitor` (Fig. 2).

5. DATA-FLOW GRAPH METRICS

Evaluating data-flow graph metrics.

The quantities introduced so far model the latencies of a single EPU. In this Section we describe how to use this information in order to compute the same metrics with respect to the whole data-flow graph. Specifically, in order to compute the metrics of a data-flow graph $G(V, E)$, it is necessary to evaluate the basic properties and metrics of each EPU of the graph. Since the quantities introduced so far are dependent on the value of the other, it is required to evaluate them following any *topological sort* of the graph G [7]. Moreover, for each EPU u in the topological sort it is necessary to evaluate the basic properties and metrics in any order that

[2]according to the convention for an empty product

Algorithm 1 output_latency(DFgraph G(V,E))

```
1: for all u in V do
2:    // Initialization.
3:    if u.isASB() then
4:       outputlat_to[u]= 0;
5:    else
6:       outputlat_to[u]= $\infty$;
7:    end if
8: end for
9: for all v in topological_sort(G) do
10:    for all u s.t. $v \in I(u)$ do
11:       // Weight of the edge.
12:       weight_vu =OL(v);
13:       // Does $v$ belong to the OL-critical path?
14:       if (u.isASB() $\wedge$ outputlat_to[u] $\leq$ outputlat_to[v] + weight_vu) $\vee$ (u.isASO() $\wedge$ outputlat_to[u] $\geq$ outputlat_to[v] + weight_vu) then
15:          // Edge contribution.
16:          outputlat_to[u] = outputlat_to[v] + weight_vu;
17:       end if
18:    end for
19: end for
20: return outputlat_to[c] + $OL(c)$;
```

satisfies the following partial ordering property: $\sigma_u(*)$ before $\rho_u(*)$ and $\rho(u)$, $\rho_u(*)$ before $OL(u)$, $OL(u)$ before $\sigma(u)$ and $AL(u)$, $AL(u)$ before $RL(u)$.

Graph Output Latency.

The overall *data-flow graph output latency*, denoted with $OL(G)$, represents the time elapsed between the **beginning** of the input event streams production by the producers and the **beginning** of the output event stream production by the consumer c.

As described in Algorithm 1, the data-flow graph output latency is determined by a set S of EPUs, $S \subseteq V$, belonging to what we call *OL-critical path.* More in detail, for each EPU v the above algorithm computes, as in line 12, the contribution of $OL(v)$ to the overall $OL(G)$: if v belongs to the OL-critical path, the contribution is taken into account (line 16). To decide whereas v belongs to the path, the rule in line 14 is applied.

Graph Activity Latency.

The overall *data-flow graph activity latency*, denoted with $AL(G)$, represents the time elapsed between the **beginning** of the input event streams production by the producers and the **end** of the output event stream production by the consumer c. Algorithm 2 illustrates how to compute the data-flow graph activity latency.

As described in Algorithm 2, the data-flow graph activity latency is determined by a set S' of EPUs, $S' \subseteq V$, belonging to what we call *AL-critical path.* Similarly to the Algorithm 1, for each EPU v the above algorithm computes, as in lines 8–12, the contribution of v to the overall $AL(G)$: if v belongs to the AL-critical path, the contribution is taken into account (line 16). To decide whereas v belongs to the path, the rule in line 14 is applied.

Graph Reactivity Latency.

The overall *data-flow graph reactivity latency*, denoted

(a) parameters and evaluation of basic properties

	$I(u)$	$n_u(*)$	$t_u(*)^3$	$n(u)$	$p(u)$	$\rho_u(*)$	$\rho(u)^4$	$\sigma_u(*)$	$\sigma(u)$
u_1	$\{\}$	$\{\}$	$\{\}$	1	y_1	$\{\}$	1	$\{\}$	y_1
u_2	$\{u_1\}$	-	$\{1\}$	1	y_2	-	1	$\{0\}$	y_2+1
u_3	$\{u_2\}$	$\{x_3\}_{x_3 \in [1,\infty)}$	-	1	y_3	$\{\frac{x_3}{x_3+(y_2+1)(x_3-1)}\}$	1	$\{y_2+1\}$	$x_3+(y_2+1)(x_3-1)+y_2$

(b) metrics evaluation

	$C(u)$	$OL(u)$	$AL(u)$	$RL(u)$
u_1	1	y_1	y_1+1	y_1
u_2	0	y_2+1	y_2+2	y_2
u_3	x_3	$y_3+x_3+(y_2+1)(x_3-1)$	$y_3+x_3+(y_2+1)(x_3-1)+1$	y_3

Table 1: EPU metrics evaluation of the data-flow graph depicted in Fig. 3 for the query `MarketDataFeedMonitor` (Fig. 2).

Algorithm 2 activity_latency(DFgraph G(V,E))

```
1: for all u in V do
2:     // Initialization.
3:     outputlenght_to[u]= $n(u)$;
4: end for
5: for all v in topological_sort(G) do
6:     for all u s.t. $v \in I(u)$ do
7:         // Weight of the edge.
8:         if u.isEB() then
9:             weight_vu =$n(v)\frac{1}{n_v(u)} \cdot n(u)$;
10:        else
11:            weight_vu =$n(v)\frac{\rho_v(u)}{t_v(u)} \cdot n(u)$;
12:        end if
13:        // Does $v$ belong to the AL-critical path?
14:        if (u.isASB() $\wedge$ outputlenght_to[u] $\geq$ outputlenght_to[v] $\cdot$ weight_vu) $\vee$ (u.isASO() $\wedge$ outputlenght_to[u] $\leq$ outputlenght_to[v] $\cdot$ weight_vu) then
15:            // Edge contribution.
16:            outputlenght_to[u] = outputlenght_to[v] $\cdot$ weight_vu;
17:        end if
18:    end for
19: end for
20: return $\frac{\text{outputlenght\_to[c]}}{\rho(c)} + OL(c)$;
```

with $RL(G)$, represents the time elapsed between the **end** of the input event streams production by the producers and the **beginning** of the output event stream production by the consumer c (see Equation 14).

$$RL(G) = OL(G) - C(G) * CHR \tag{14}$$

Graph Complexity.

The overall *data-flow graph complexity*, denoted with $C(G)$, represents the *average* number of events that are to be produced by the producers in order to make the consumer c produce at least $n(c)$ events, with respect to the producer v that represents the worst case, i.e., that maximizes this quantity. Algorithm 3 illustrates how to compute the data-flow graph complexity.

As described in Algorithm 3, the data-flow graph activity latency is determined by a set S'' of EPUs, $S'' \subseteq V$, belonging to what we call *C-critical path.* Similarly to the

Algorithm 3 complexity(DFgraph G(V,E))

```
1: for all u in V do
2:     // Initialization.
3:     complexity_to[u]= 0;
4: end for
5: for all v in topological_sort(G) do
6:     for all u s.t. $v \in I(u)$ do
7:         // Weight of the edge.
8:         weight_vu = $\frac{C(u)}{n(v)}$;
9:         // Does $v$ belong to the C-critical path?
10:        if complexity_to[u] $\leq$ complexity_to[v] $\cdot$ weight_vu then
11:            // Edge Contribution.
12:            complexity_to[u] = complexity_to[v] $\cdot$ weight_vu;
13:        end if
14:    end for
15: end for
16: return complexity_to[c];
```

Algorithms 1 and 2, for each EPU v the above algorithm computes, as in lines 8, the contribution of v to the overall $C(G)$: if v belongs to the C-critical path, the contribution is taken into account (line 12). To decide whereas v belongs to the path, the rule in line 10 is applied.

Fig. 6 and Tab. 2 show, respectively, the timing properties and the evaluation of the metrics of the data-flow graph of Fig. 3, for the query `MarketDataFeedMonitor` (Fig. 2).

6. CONCLUSION

The paper presents a formal model to evaluate some cost metrics of a continuous streaming computation, represented as a data-flow query graph where each node is a basic query - namely EPU event processing unit - incoming edges to a node represent input streams and outcoming edges are output streams from a node. Each EPU is a generic unit that performs a computation on input streams. The model is able to associate several latencies metrics with a data-flow graph as well as compute the complexity of the input necessary to produce an event as output (data flow graph complexity).

The model can be used during the process of designing a solution of a problem expressed in terms of continuous query computation, in order to evaluate off-line the data flow graph that better fits the QoS requirements of a programmer in

	$C(u)$	$OL(u)$	$AL(u)$	$RL(u)$
	0	$y_1 + y_3 + (2 + y_2)x_3$	$y_1 + y_3 + (2 + y_2)x_3 + 1$	$y_1 + y_3 + (2 + y_2)x_3$

Table 2: Metrics evaluation of the data-flow graph of Fig. 3 for the query `MarketDataFeedMonitor` (Fig. 2).

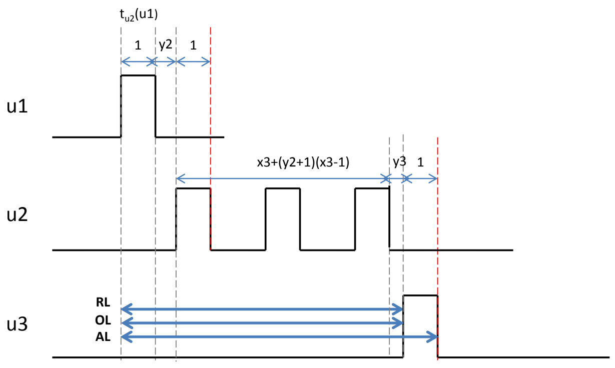


Figure 6: Timing diagram of the data-flow graph of Fig. 3 for the query `MarketDataFeedMonitor` (Fig. 2). See 1 for EPU details.

terms of latency and data flow graph complexity. The model works under the assumption that the processing time at a node is much shorter than the waiting time for input data. This is an assumption that is practically verified in many specific applicative context such as command and control application, collaborative security etc. where systems are built in order to have low resource contention.

Our plans to enhance the work focus on different aspects. From the model point of view, we are extending it in order to represent EPU parameters as a function of time and model the dynamic behavior of the query data-flow graph. Moreover we are introducing in the model constraints on the available resources.

On the practical side, we have recently implemented a preliminary version of a software tool that is able to compute the metrics introduced in this paper starting from a description of the query data-flow graph. The current version of the tool is implemented in Python and works as follows. The programmer specifies through XML files a data-flow graph that may resolve a specific query expressed in natural language. The XML files include for each EPU of the processing system: (i) the EPU parameters we introduced; (ii) the different input of the EPU; (iii) the processing behavior of the EPU, that is, all-stream batch or all-stream online processing, and time-based or event-based processing. The tool is able to visualize the data-flow graph and compute - with respect to both each graph node and entire data-flow graph - our metrics of interest. This allows the programmer to select the best data-flow graph that can be submitted to a CEP or DBMS according to the Quality of Service requirements to be met.

7. ACKNOWLEDGEMENTS

This work has been partially supported by the EU project CoMiFin on the protection of the Financial Infrastructure from Cyber Attacks.

8. REFERENCES

[1] Where Complex Event Processing meets Open Source: Esper and NEsper. `http://esper.codehaus.org/`, 2009.

[2] System S. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html`, 2010.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[4] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, 2004.

[5] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *Proceedings of the 9th international workshop on Data Bases Programming Languages (DBPL '03)*, Lecture Notes in Computer Science, pages 1–19, 2003.

[6] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *Proceedings of the 27st IEEE International Conference on Data Engineering*, ICDE '05, pages 255 –266, 2011.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[8] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. 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, 2008.

[9] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proceedings of the 25th international Conference on Information and Knowledge Management*, CIKM '03, pages 271–278, 2003.

[10] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of Data*, SIGMOD '02, pages 37–48, 2002.

[11] Y. Xing. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st IEEE International Conference on Data Engineering*, ICDE '05, pages 791–802, 2005.

[12] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International conference on Very Large Data Bases*, VLDB '06, pages 775–786, 2006.