Probabilistic Management of Late Arrival of Events

Nicolo Rivetti
Technion – Israel Institute of Technology
Haifa, Israel
nrivetti@technion.ac.il

Nikos Zacheilas
Athens University of Economics and Business
Athens, Greece
zacheilas@aueb.gr

Avigdor Gal
Technion – Israel Institute of Technology
Haifa, Israel
avigal@technion.ac.il

Vana Kalogeraki
Athens University of Economics and Business
Athens, Greece
vana@aueb.gr

ABSTRACT

In a networked world, events are transmitted from multiple distributed sources into CEP systems, where events are related to one another along multiple dimensions, *e.g.*, temporal and spatial, to create complex events. The big data era brought with it an increase in the scale and frequency of event reporting. Internet of Things adds another layer of complexity with multiple, continuously changing event sources, not all of which are perfectly reliable, often suffering from late arrivals. In this work we propose a probabilistic model to deal with the problem of reduced reliability of event arrival time. We use statistical theories to fit the distributions of inter-generation at the source and network delays per event type. Equipped with these distributions we propose a predictive method for determining whether an event belonging to a window has yet to arrive. Given some user-defined tolerance levels (on quality and timeliness), we propose an algorithm for dynamically determining the amount of time a complex event time-window should remain open. Using a thorough empirical analysis, we compare the proposed algorithm against state-of-the-art mechanisms for delayed arrival of events and show the superiority of our proposed method.

CCS CONCEPTS

• **Information systems → Stream management**;

KEYWORDS

Complex Event Processing, Sliding Window, Late arrivals, Probabilistic Prediction

ACM Reference Format:
Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. 2018. Probabilistic Management of Late Arrival of Events. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210284.3210293

1 INTRODUCTION

In a networked world, events are transmitted from multiple distributed sources into complex event processing (CEP) systems, where events are related to one another along multiple dimensions, *e.g.*, temporal and spatial, to create complex events. The big data era brought with it an increase in scale in the number of reported events (*volume*), frequency of reporting (*velocity*), and number of event sources (*variety*). The Internet of Things adds another layer of complexity with multiple, continuously changing event sources, not all of which are perfectly reliable (*veracity*).

A use-case in point is that of smart cities [2], where sensors from people, vehicles, CCTV monitors, *etc.* are combined, processed and emitted to multiple systems (possibly located in different cities) for assisting urban area residents, visitors, and authorities to identify traffic congestions, prevent criminal activities, and facilitate a shared economy.

Of particular interest to us in this work is the reliability of event arrival time. The combination of unreliable event sources with unreliable networks may create long delays in reporting events. Such delays are not identical across the board. To start with, some sensors are more reliable than others. Also, events may arrive from different network regions and may suffer from varying levels of congestion. Finally, priorities in different middleware agents en route may cause some events to accumulate more delays than others.

Late arrival of events affect the accuracy and efficiency of the CEP systems. Complex events are monitored using time windows, which are triggered by the arrival of some event [3]. Such windows are kept open for an amount of time as determined by the complex event definition and may be closing too soon, before the (late) arrival of some missing events. Waiting longer for missing events (that may never show up) exhausts system resources and affects the promptness of event reporting. Therefore, it is necessary to provide mechanisms to decide when a time window must be closed.

An effective assessment regarding the time window closing should be based on event inter-generation delay as well as the expected network delay. Assessing these measures may not be a trivial task, as illustrated in Figure 1. The figure demonstrates the variance in event inter-generation delay and network delay, as captured by a real-world traffic monitoring application. Bus sensor data are used to detect congestion in different areas of a city. The sensor data are initially gathered by a concentrator component and then forwarded to a CEP component, deployed in a remote

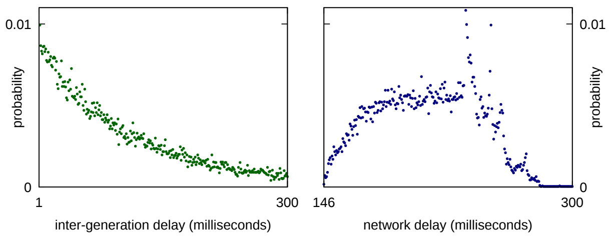

Figure 1: Event inter-generation and network delays.

location (see Section 4.2 for a detailed description of the application). Figure 1 (left) demonstrates the impact of concentration on event inter-generation delay. While the distribution resembles a power-law distribution, the tail is thick, with multiple events (with probability higher than 3×10^{-3}) being published with an inter-generation delay larger than 80 milliseconds. Figure 1 (right) depicts the frequency distribution of network delay between the concentrator and the CEP system, demonstrating a non-trivial distribution.

In this work we propose a probabilistic framework for analyzing late arrival of events. We use statistical theories to fit the relevant delay distributions per event type. Equipped with these distributions and some user-defined tolerance levels, we propose an algorithm for dynamically determining the amount of time a complex event time window should remain open. We also propose a predictive method for determining whether a complex event has occurred, in the absence of late arriving events.

Late event arrival was discussed in the literature (see detailed discussion in Section 5). Closest to our work are [6, 7, 21], all differ in that they focus on specific operator types (*e.g.,* either join [6] or aggregate [7] operators) and assume that event inter-generation time is deterministically fixed while our technique is operator agnostic and models event inter-generation time in probabilistic terms. To the best of our knowledge, our work is the most comprehensive in its probabilistic modeling, allowing a fine granularity analysis of late arrival time from multiple event sources and providing maximum flexibility with respect to user's preferences. Our contribution is four-fold:

- We propose a probabilistic model of temporal delay in event reporting.
- We propose an algorithm (ProbSlack) to determine the remaining time before reaching a decision about a complex event. The algorithm is enhanced for performance via efficient caching.
- We present a thorough empirical analysis using both synthetic and real-world datasets. We compared ProbSlack against state-of-the-art mechanisms for delayed arrival of events and show the superiority of our probabilistic modeling approach.
- Finally, we provide a prototype implementation, based on Apache Flink [17] (Section 3.2), and a new dataset (Section 4.2) consisting of a stream of events collected in a smart city application, with their generation and reception timestamps.

The rest of the paper is organized as follows. Section 2 presents a model and sets the problem definition in its context. An algorithm and several efficiency enhancement mechanisms are presented in Section 3. Empirical evaluation follows in Section 4. We conclude the paper with related work (Section 5) and concluding remarks (Section 6).

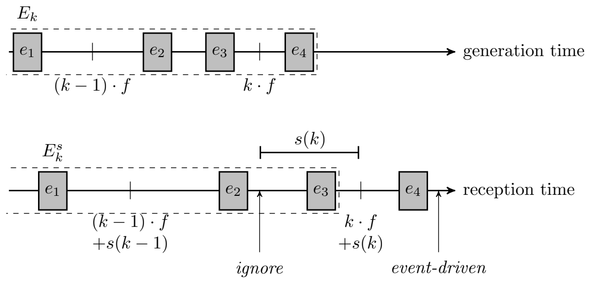

Figure 2: Problem and notation.

2 MODEL AND PROBLEM STATEMENT

A data stream $\sigma = \langle e_1, e_2, \ldots \rangle$ contains a (possibly infinite) sequence of *events*, where events of similar structure and meaning belong to the same event type, denoted by $e.type$. An event is associated with a set of attributes, dictated by their type. In common to all events is an attribute e_{id}, which uniquely identifies the event and a type as detailed above. We use a *point-based semantics*, such that an event e is associated with one or more timestamp attributes. In this work, we associate an event with two timestamps, namely a generation timestamp $e.gts$ and a reception timestamp $e.rts$. The former represents the timestamp in which an event was generated at the source, while the reception timestamp is the time at which the event was received by the CEP engine. In the context of pub/sub systems, generation and reception times may be conceived as published time and the time an event reaches a subscriber [11].

We assume that events leave a source and are received ordered according to their generation time and that for any event e, $e.gts \leq e.rts$. Otherwise, we make no assumptions on the synchrony of event sources and engine clocks.

Of particular interest to us are sliding window queries. Given a finite epoch $\tau = \{t_0, t_1, \ldots, t_n\}$ of n timepoints, a sliding window query is a triple $SWQ = \langle f, l, p \rangle$, where f stands for the frequency of the query, l stands for the length of the sliding window, and p specifies the required event pattern we would like to identify. For the sake of clarity and without loss of generality, we assume that n is a multiple of f ($n \mod f = 0$).

Example 2.1. Given an epoch of 100 seconds and an event stream σ with events types c_1, c_2, the sliding window query $SWQ = \langle 10, 5, c_1\ SEQ\ c_2 \rangle$ checks every 10 seconds if the past 5 seconds contained an event of type c_1 followed by an event of type c_2.

The set of relevant events from a stream σ in the k-th sliding window ($1 \leq k \leq \frac{n}{f}$) of a query $SWQ = \langle f, l, p \rangle$ in an epoch τ is defined as follows $E_k = \{e \in \sigma | k \cdot f - l \leq e.gts \leq k \cdot f\}$ (Figure 2, top). The presence of transmission delays may introduce erroneous response to a sliding window query. In particular, if we evaluate SWQ in the most straightforward way, *i.e.*, at time points $f, 2 \cdot f, \ldots, n$ we are bound to miss events whose generation time is within the window while their reception time is beyond that window. To resolve the matter, one may consider rewriting the original query by separating the size of the window from its evaluation interval. Therefore, we propose a rewritten sliding window query $RSWQ = \langle f, s(k), l, p\rangle$ where f, l and p are defined as before and $s(k)$ is a slack function, which determines the amount of time an evaluation of the query is postponed. After rewriting, evaluating the k-th window $[k \cdot f - l, k \cdot f]$ is performed at time $k \cdot f + s(k)$ (Figure 2, bottom).

A common approach to handle generation time and the transmission delays in sliding window queries is through *punctuation* [10] (or *watermarking*). Periodically and/or for each received event, the engine injects a system event carrying a generation time value that is assumed to be in the past. While this technique provides an elegant framework, the user (system administrator or end-user) still has to define a function that, given the current state of the engine, generates the correct generation time. There are in general three possible policies (Figure 2, bottom) to handle late arrivals (due to, *e.g.*, jitter in the communication link) in sliding window queries:

- *Ignore* — according to this policy we set $\forall k, s(k) = 0$ and therefore *RSWQ* effectively becomes *SWQ*. Such a policy ensures a timely, yet possibly inaccurate response.
- *Event-driven* — The window is kept open until an event e such that $e.gts > k \cdot f$, for the k-th window, arrives. This policy guarantees that all relevant events are processed at the expense of timeliness.
- *User-defined* — this policy makes use of a user-defined latency tolerance parameter u and sets $\forall k, s(k) = u$. The user may be able to strike a good balance between timeliness and quality, however this in general requires a deep understanding of the delay properties. A common value for parameter u is the average network delay [15].

We propose a fourth policy, *probability-based*, according to which function s is based on a stochastic analysis of the inter-generation and transmission delays. Any such policy should balance the trade-off between two contradicting goals. The first is the ability to correctly identify patterns in sliding window queries. We denote by $E_k^s = \{e \in \sigma | k \cdot f - l \le e.gts \le k \cdot f \wedge e.rts \le k \cdot f + s(k)\}$ the set of captured events from a stream σ in the k-th sliding window ($1 \le k \le \frac{n}{f}$) of a query $SWQ = \langle f, l, p\rangle$ in an epoch τ. These are all the events that are relevant to the query and whose reception time falls within the time window including the extra slack time $s(k)$. The indicator

$$\mathbb{I}_k = 1 \text{ if } E_k \setminus E_k^s \neq \emptyset$$

specifies whether a window missed relevant events and the average number of missed patterns is computed to be

$$MER(RSWQ) = \frac{f}{n} \sum_{k=1}^{\frac{n}{f}} \mathbb{I}_k \tag{1}$$

The second goal relates to the latency in closing a window, which is computed over the rewritten query *RSWQ* as follows

$$SL(RSWQ) = \sum_{k=1}^{\frac{n}{f}} s(k) \tag{2}$$

The problem we aim at solving is bi-objective, minimizing simultaneously both *MER* and *SL*. There are various approaches to solve such bi-objective problems which include the identification of a Pareto curve of solutions and setting priorities among the objectives [20]. We propose the following problem definition, aiming at minimizing the *slack latency SL* while keeping *missed event ratio MER* within a user-defined budget. It is worth noting that the optimization focuses on the slack parameter $s(k)$, and therefore we use the rewritten query *RSWQ* rather than the original query *SWQ*.

Listing 1: Sliding window TRIGGER function.

```
1: function TRIGGER($e$)                      ▷ Last received event
2:     $t \leftarrow t + 1$
3:     while COND($k, t, e$) do $k \leftarrow k + 1$
4:     return $k - 1$   ▷ All windows $k' \le k - 1$ can be processed
```

Listing 2: *Ignore* policy COND function.

```
1: function COND($k, t, e$)
2:     return $t \ge k \cdot f$
```

PROBLEM 2.2 (WINDOW SLACK OPTIMIZATION). *Given a sliding window query $SWQ = \langle f, l, p\rangle$ and a user tolerance to windows with missed events δ, the* Window Slack Optimization *aims at finding a rewriting of SWQ to $RSWQ = \langle f, s(k), l, p\rangle$ s.t.*

$$\min \; SL(RSWQ) \tag{3}$$

$$\textit{subject to } MER(RSWQ) \le \delta \tag{4}$$

It is worth noting that Problem 2.2 depends solely on $s(k)$, which in turn depends on event arrival frequency, network delays, *etc.* In what follows we offer a generic solution that may be applied to any number of different event types.

3 METHODOLOGY

A common implementation of a sliding window is through a TRIGGER function, a function that is called at each timepoint to evaluate whether, given the current state of the system, the sliding window is ready to be processed [3]. To solve Problem 2.2, the TRIGGER function has to evaluate whether there are missing events before the window can be safely processed. Depending on the policy of choice (see Section 2) the TRIGGER function has to choose whether to process the window as it is (using some of the *MER* budget), or to wait (increasing *SL*).

Listing 1 presents a possible implementation of the TRIGGER function. The while loop (line 3) scans all the window identifiers that can be closed at time t. The function COND encapsulates the condition of the applied policy.

For the k-th sliding window ($1 \le k \le \frac{n}{f}$) of a query $SWQ = \langle f, l, p\rangle$ in an epoch τ over a stream σ, the core problem is to accurately evaluate for each timepoint $t \ge k \cdot l - f$ whether

$$\nexists e_i \in E_k \text{ such that } e_i.rts > t \tag{5}$$

In practice, it is possible to evaluate Eq. 5 deterministically only in rare cases, under strict assumptions. Applying the *ignore* policy (see Listing 2 for the implementation of function COND for the *ignore* policy) may indiscriminately violate Constraint 4. In contrast, the *event-driven* policy leads to large *SL*, which is not aligned with the minimization requirement of Eq. 4. User-defined solutions fail to provide any guarantees on either *SL* or *MER*. In this section we propose ProbSlack, a probabilist framework to evaluate Eq. 5.

Listing 3: ProbSlack COND function.

```
function COND($k, t, e$)
    if $e \neq e_{prev}$ then
        UPDATEFITTING($e.gts - e_{prev}.gts, e.rts - e.gts$)
        $e_{prev} \leftarrow e$
        if $\lceil \frac{e.gts}{f} \rceil < k \wedge k \neq k^{\text{last}}$ then
            $ME \leftarrow ME + min\{k - k^{\text{last}}, k - \lceil \frac{e.gts}{f} \rceil\}$
            $k^{\text{last}} \leftarrow k$
    $c \leftarrow e.gts > k \cdot f$
    if $\frac{ME}{k-1} \geq \delta$ then RESETFITTING($t$)
    else $c \leftarrow c \vee \neg P_{\text{miss}}(k, t, e)$
    return $c \wedge t > (k-1) \cdot f$
```

ProbSlack drives the TRIGGER function decision making towards reduced *SL* while satisfying the *MER* bound in expectation.

3.1 ProbSlack

To satisfy Constraint 4 in expectation, we wish to compute first the probability that Eq. 5 is not satisfied at time t, as follows:

$$\mathbb{P}_{\text{miss}}(k, t) = \mathbb{P}[\exists e_i \in E_k \text{ s.t. } e_i.rts > t] \tag{6}$$

Listing 3 shows the implementation of the COND function for ProbSlack. For the sake of clarity, we will initially ignore lines 2- 7 and line 9 . The core of our approach is in line 10. A window is ready to be processed if the probability that an event may be missed is below the user defined threshold ($\mathbb{P}_{\text{miss}}(k, t) \leq \delta$) using an indicator $P_{\text{miss}}(k, t, e)$ that is set to true if $\mathbb{P}_{\text{miss}}(k, t) > \delta$. This expression is OR-ed with the *event-driven* policy (line 8), indicating that if Eq. 5 is deterministically satisfied the window can be processed. As an example, considering Figure 2 and let k be the current window. If the engine receives event e_4 before reception time $k \cdot f + s(k)$ it can safely close the window k. Given sufficient evidence, ProbSlack may close the $k-1$ window even before $k \cdot f$. Using Figure 2 once more, window $k-1$ may have been closed just after the arrival of event e_1 and thus before time $(k-1) \cdot f$, introducing a noticeable gain in *SL* without harming *MER*. On the other hand, the user expects that by time $(k-1) \cdot f$ at most $k-1$ windows are closed so no windows are processed ahead of time. This condition is enforced by AND-ing the disjunction with the expression $t > (k-1) \cdot f$ (line 11).

THEOREM 3.1 (PROBSLACK CORRECTNESS). *The output of ProbSlack satisfies in expectation Constraint 4:*

$$\mathbb{E}[\delta - MER] \geq 0$$

PROOF. By construction, $\mathbb{P}_{\text{miss}}$ models the probability that Eq. 6 is satisfied. It is equal to the probability that the indicator $\mathbb{I}_k$ is satisfied:

$$\begin{aligned}\mathbb{P}_{\text{miss}}(k, t) &= \mathbb{P}[\exists e_i \in E_k \text{ s.t. } e_i.rts > t] \\ &= \mathbb{P}[E_k \setminus E_k^s \neq \emptyset] = \mathbb{P}[\mathbb{I}_k = 1]\end{aligned}$$

Consider $\mathbb{E}[\delta - MER]$, the expected value of the difference between the budget δ and the value of *MER*. By line 10 in Listing 3 and linearity of expectation we have

$$\begin{aligned}\mathbb{E}[\delta - MER] = \mathbb{E}[\delta] - \mathbb{E}[MER] &= \delta - \frac{f}{n} \sum_{k=1}^{\frac{n}{f}} \mathbb{E}[\mathbb{I}_k] \\ &= \delta - \mathbb{E}[\mathbb{I}_k] = \delta - \mathbb{P}[\mathbb{I}_k = 1] \text{ (indicator properties)} \\ &= \delta - \mathbb{P}_{\text{miss}}(k, t) \geq 0 \text{ (since } \mathbb{P}_{\text{miss}}(k, t) \leq \delta),\end{aligned}$$

which concludes the proof. □

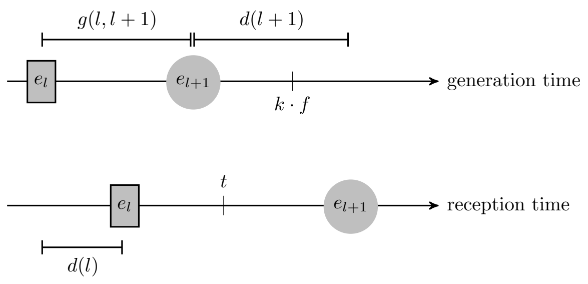

Figure 3: ProbSlack modeling.

Consider the variance $\text{Var}[\delta - MER]$. By construction, $MER \in [0, 1]$, thus $\mathbb{E}[MER^2] \leq \mathbb{E}[MER]$ by monotonicity of the expectation, then

$$\text{Var}[\delta - MER] = \text{Var}[MER] = \mathbb{P}_{\text{miss}}(k, t) - \mathbb{P}_{\text{miss}}(k, t)^2 \leq \delta - \delta^2 \leq \frac{1}{4}$$

The variance is maximized in $\mathbb{P}_{\text{miss}}(k, t) = 0.5$. For values of δ other than 0.5, the variance is bounded by $\delta - \delta^2 < 0.25$, *e.g.*, for $\delta = 0.2$ the variance is bounded by 0.16.

We can introduce a scaling factor ε such that $\mathbb{P}_{\text{miss}}(k, t) \leq \varepsilon\delta$. Using Markov's inequality we can arbitrarily bound the probability that $MER > \delta$, as follows.

$$\mathbb{P}[MER > \delta] \leq \frac{\mathbb{E}[MER]}{\delta} = \frac{\mathbb{P}_{\text{miss}}(k, t)}{\delta} \leq \frac{\varepsilon\delta}{\delta} = \varepsilon.$$

Due to the coarseness of the bound, the scaling ε factor may cause a large increase of *SL*. To avoid such undesired increase, we introduce a guarding mechanism to avoid trajectories leading to a violation of Constraint 4. In particular, we track the value of *MER* (lines 5 and 7 of Listing 3) and revert to the more conservative *event-driven* policy if $MER \geq \delta$. In other words, if $MER \geq \delta$ then function P_{miss} is not evaluated (lines 9 and 10) and only the *event-driven* condition $e.gts > k \cdot f$ is considered (line 8).

A final note regarding ProbSlack is that it supports multiple event types, applying the same approach for each type separately.

$\mathbb{P}_{\text{miss}}$ computation — To evaluate $\mathbb{P}_{\text{miss}}(k, t)$ we first model both the generation and reception time of a yet to be received event. Figure 3 illustrates our approach, where $\mathbb{P}_{\text{miss}}(k, t)$ can be represented as the probability that the last event received by the system in the current window k, $e_l \in E_k$, is followed by at least one other event $e_{l+1} \in E_k$ that is yet to be received, *i.e.*, $e_{l+1}.rts > t$. Let

- $g(i, i+1) \triangleq e_{i+1}.gts - e_i.gts$ be the inter-generation time between two consecutive events,
- $d(i) \triangleq e_i.rts - e_i.gts$ be the network delay of event e_i, and

- $e_l \in E_k$ ($e_l.rts < t$) be the last received event in the window.

g and d are random variables. It is worth noting that g and d are independent,*i.e.*, g models the behavior of the event generator and d models the network characteristics. $\mathbb{P}_{\text{miss}}(k,t)$ can be expressed using g and d, as follows:

$$\begin{aligned}\mathbb{P}_{\text{miss}}(k,t) &= \mathbb{P}[e_{l+1}.gts \leq k \cdot f \wedge t < e_{l+1}.rts] \quad (7)\\ &= \int_0^{k \cdot f - e_l.gts} pdf_g(x) \int_{t-e_l.gts}^{+\infty} pdf_d(y+x) dy dx,\end{aligned}$$

Then, the computation of $\mathbb{P}_{\text{miss}}$ boils down to computing the pdf_{g} of g and the cdf_{d} of d.

On-line fitting of g and d — Fitting a distribution is an extensively studied research area in the data mining community. There are two main methods, namely parametric and non-parametric fitting [13]. The former entails choosing a well defined parametric probability distribution and tuning its parameters to fit the data distribution. It is extremely compact, with only the parameters and *pdf* function to be stored, but the quality of the fitting completely depends on whether the chosen distribution family is able to adjust to the data distribution shape. Non-parametric fitting accommodates any data distribution at the expenses of a larger memory footprint. More specifically, the basic method for non-parametric fitting is to build a histogram of the data distribution. Then, the number of histogram bins needed to accurately fit the data distribution may be large. Non-parametric fitting of a continuous distribution also introduces a discretization error.

We argue that non-parametric fitting is well suited for our use-case. First, source and network behavior may be too erratic to be captured by a parametric distribution (*cf.*, Figure 1). Second, our scenarios allow straightforward discretization of time and a manageable bounded number of bins. Specifically, the clocks of CEP engines and event publishers, as most IT systems, have a finite resolution that is seldom below milliseconds. This practically means that event generation and reception timestamps is done in discrete time and discretizing g and d distributions does not introduce any additional error. Notice that we can then replace the integral of Eq. 7 with a summation. Moreover, the user's and system's time requirements are given in some discrete finite interval, say $\{0, \rho, 2\rho, \ldots, \omega\rho\}$, where the step ρ is a multiple of clock resolution. We argue that the number of distinct ω values in this interval is reasonable (*e.g.*, hundreds), which leads to a bounded and reasonable number of bins for exact frequency distribution fitting of g and d.

To handle distribution drift in g and d, we maintain two key-value sets, $\mathfrak{F}_g$ and $\mathfrak{F}_d$, both being continuously trained (line 3 of Listing 3 and lines 12 to 16 of Listing 4). Let T be a user-defined period. We assume that T is sufficiently large to let the histogram capture the distribution, yet sufficiently small to react to changes in a timely manner. At each period end (lines 16 and 18), both sets are reset (line 22). Abrupt distribution changes may cause bias in the evaluation of $\mathbb{P}_{\text{miss}}$ for up to $2T$ events yet the guarding mechanism introduced in Listing 3 prevents ProbSlack from exceeding the *MER* budged δ. Moreover, when *MER* reaches δ, the fitting is reset (line 9).

In use-cases where the distributions of g and d continuously change, ProbSlack may benefit from a sliding window fitting. Such extension can be implemented using approximations of frequency distributions over sliding window with sub-linear space complexity [12]. These techniques can also be used to handle extreme scenarios where the number of distinct ω values is too large.

Listing 4: Jumping window fitting.

```
12: function UpdateFitting($g, d$)
13:     $\mathfrak{F}_g \leftarrow \langle g, 1+ \text{ value of } \langle g, v_g \rangle \in \mathfrak{F}_g \text{ or } 0 \rangle$
14:     $\mathfrak{F}_d \leftarrow \langle d, 1+ \text{ value of } \langle d, v_d \rangle \in \mathfrak{F}_d \text{ or } 0 \rangle$
15:     $m \leftarrow m + 1$
16:     ResetFitting($t_{prev} + m$)
17: function ResetFitting($t$)
18:     if $t \geq t_{prev} + T$ then
19:         $t_{prev} \leftarrow t$
20:         $m \leftarrow 0$
21:         BuildCache()
22:         $\mathfrak{F}_g, \mathfrak{F}_d \leftarrow \emptyset$
```

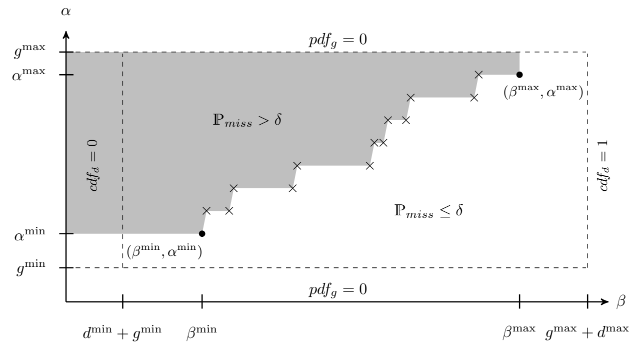

Figure 4: Caching graphical representation.

$\mathbb{P}_{\text{miss}}$ efficient computation — Computing

$$\mathbb{P}_{\text{miss}}(k,t) = \sum_{g=0}^{\alpha} pdf_g(g)(1 - cdf_d(\beta - g)),$$

where $\alpha = k \cdot f - e_l.gts$ and $\beta = t - e_l.gts$, using directly the key-value sets $\mathfrak{F}_g$ and $\mathfrak{F}_d$ may induce a large computational cost. In the worst case each evaluation of $\mathbb{P}_{\text{miss}}$ generates $O(N_\alpha)$ lookups, where N_α is the number of distinct values of α. To avoid redundant computations we introduce a caching mechanism (see Listing 5) to store pre-computed values of $\mathbb{P}_{\text{miss}}$ for all possible parameter combinations, reducing the query time complexity to $O(1)$ (*cf.*, lines 23 to 25 in Listing 5). Let $[g^{\min}, g^{\max}]$ and $[d^{\min}, d^{\max}]$ be the supports of g and d, respectively. Then, for $g \notin [g^{\min}, g^{\max}]$ and/or $\beta > d^{\max} + g^{\max}$, $pdf_g(g)(1 - cdf_d(\beta - g)) = 0$ (*cf.*, Figure 4). Therefore, $\mathbb{P}_{\text{miss}} = 0$ for value of $\beta > d^{\max} + g^{\max}$, and the value of $\mathbb{P}_{\text{miss}}$ does not increase for values of $\alpha \notin [d^{\min}, d^{\max}]$. Furthermore, we do not need to store the value of $\mathbb{P}_{\text{miss}}$ for all other possible combination of β and α. In particular we only need to find the coordinates of the points (crosses in Figure 4) forming the poly-line from $(\beta^{\min}, \alpha^{\min})$ to $(\beta^{\max}, \alpha^{\max})$. This poly-line defines the area (grayed in Figure 4) where $\mathbb{P}_{\text{miss}} > \delta$.

To build the cache, ProbSlack first identifies $\alpha^{\min}$ and $\beta^{\min}$ (line 28 and 29). Notice that, $\beta^{\min}$ can be found through dichotomy

Listing 5: Cached $\mathbb{P}_{\text{miss}}$ evaluation

```
23: function $P_{\text{miss}}(k, t, e_l)$
24:     $\beta^* \leftarrow t - e_l.gts$; $\alpha^* \leftarrow k \cdot f - e_l.gts$
25:     return $\exists\langle\beta^-, \beta^+, \widehat{\alpha}\rangle \in C$ s.t. $\beta^* \in [\beta^-, \beta^+] \wedge \alpha^* \geq \widehat{\alpha}$
26: function BuildCache()
27:     $\forall\langle d, v_d\rangle \in \mathfrak{F}_d, \mathfrak{F}_d \leftarrow \langle d, \sum_{\forall d' < d} v_{d'}\rangle$
28:     $\alpha^{\min} \leftarrow \min \alpha$ s.t. $\sum_{g=g^{\min}}^{\alpha} \text{PDF}_g(g) > \delta$
29:     $\beta^{\min} \leftarrow \max \beta$ s.t. $\sum_{g=g^{\min}}^{\alpha^{\min}} \text{CP}(g, \beta) > \delta$
30:     $C \leftarrow C \cup \{\langle -\infty, \beta^{\min}, \alpha^{\min}\rangle\}$
31:     for $\beta \in ]\beta^{\min}, d^{\max} + g^{\max}]$ do
32:         $\widehat{\alpha} \leftarrow \min \alpha$ s.t. $\sum_{g=g^{\min}}^{\alpha} \text{CP}(g, \beta) > \delta$
33:         if $\widehat{\alpha} > g^{\max}$ then break
34:         if $\nexists\langle\beta^-, \beta - 1, \widehat{\alpha}\rangle$ then $C \leftarrow C \cup \{\langle\beta, \beta, \widehat{\alpha}\rangle\}$
35:         else $C \leftarrow C \setminus \langle\beta^-, \beta - 1, \widehat{\alpha}\rangle \cup \{\langle\beta^-, \beta, \widehat{\alpha}\rangle\}$
36: function CP$(g, \beta)$
37:     if $\beta - g < d^{\min}$ then return $\text{PDF}_g(g)$
38:     return $\text{PDF}_g(g)\ \text{CDF}_d(\beta - g)$
39: function $\text{PDF}_g(d)$
40:     return $v_g/T$ of $\langle g, v_g\rangle \in \mathfrak{F}_g$
41: function $\text{CDF}_d(d)$
42:     return $v_d/T$ of $\langle d, v_d\rangle \in \mathfrak{F}_d$
```

search in a logarithmic number of steps. For all values of $\beta \in]\beta^{\min}, d^{\max} + g^{\max}]$ (line 31), ProbSlack identifies the minimum value of α such that $\mathbb{P}_{\text{miss}} > \delta$ (line 32) for the given β, until $\mathbb{P}_{\text{miss}} > \delta$ cannot be satisfied anymore (line 33). These cached values (lines 34 to 35) are the coordinates of the poly-line points in Figure 4.

Theorem 3.2 (ProbSlack Space and Time Complexities). *Let N_α and N_β be the number of distinct values of α and β, and let $T > N_\alpha \times N_\beta$. Then, ProbSlack's amortized query and update complexity are $O(1)$. ProbSlack worst case space complexity is $O(\max\{N_\alpha, N_\beta\})$.*

Proof. By Listing 1 each call of the trigger function entails the evaluation of the cond function and may execute the while loop (line 3). By Listing 3 and the caching mechanism detailed in Listing 5, function cond complexity is $O(1)$. A single call of the trigger function can evaluate the cond function from one to $\frac{n}{f} + 1$ times, *i.e.*, the worst case query complexity of ProbSlack is $O(\frac{n}{f} + 1)$. However, over all n timepoints, function cond is called at most $\frac{n}{f} + n$ times, *i.e.*, the amortized query time complexity is $O(1)$.

By Listing 4, each new event arrival causes 2 dictionaries to be updated. Furthermore, the $\mathbb{P}_{\text{miss}}$ cache may be rebuilt. By Listing 5, rebuilding the cache has $O(N_\alpha \times N_\beta)$ time complexity. However, the cache is rebuilt each $T > N_\alpha \times N_\beta$ events. Then the amortized update time complexity is $O(1)$.

Finally, at all times ProbSlack maintains 2 dictionaries of size $O(\max\{N_\alpha, N_\beta\})$. The cache stores at most the coordinates of the points of the poly-line in Figure 4, which requires $3 \times N_\alpha$ integers. Therefore, the worst case space complexity is $O(\max\{N_\alpha, N_\beta\})$. □

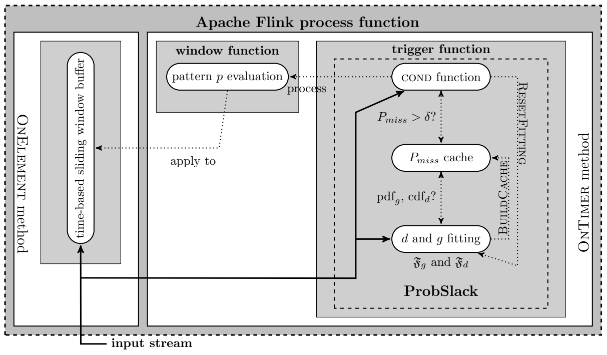

Figure 5: ProbSlack prototype architecture

In practice, the cache is more memory and time efficient. For instance, in our experiments the cache had a maximum size of 7 entries (*cf.*, Section 4.3). Moreover, ProbSlack uses a lazy version of the caching algorithm, further reducing the resource footprint. In particular, lazy caching prevents the evaluation of β values that are never observed and spreads the cache building over time.

3.2 Prototype

ProbSlack expects to have full control on when a window is processed and to have access to the generation and reception timestamps of any event received by the operator. These are features that can be easily implemented around most time-based window operators, therefore ProbSlack can be plugged into most Stream and Complex Event Processing engines with an implementation of the time-based window operator.

Apache Flink [17] is one of the few open-source CEP engines, with a large deployment based in both academia and industry, handling natively event generation time. It can naturally handle the implementation of several of our baselines, making it the framework of choice for our implementation. Furthermore, its highly pluggable architecture made the prototype development straightforward.

ProbSlack[1] is implemented through Flink's (version 1.4.0) process function,[2] *i.e.*, keyBy(*busId*).process(new ProbSlackClass()). The ProbSlackClass() overrides the ProcessFunction#processElement method buffering the received events. The overridden ProcessFunction#onTimer method is called at each timepoint and implements ProbSlack logic as well as the $SWQ = \langle f, l, p\rangle$ query. In Figure 5 we illustrate the prototype architecture of ProbSlack on top of Apache Flink.

3.3 Extensions

We next discuss briefly two variants of Problem 2.2.

Maximum allowed latency — While the user is interested in enforcing a minimum level of required quality (*i.e.*, bounding *MER*), she (or the system administrator) may also require that the processing of a window cannot be delayed for too long. This is, in general, modeled by introducing a *maximum allowed latency* (*mal*) parameter bounding $s(k) \leq mal$. ProbSlack can incorporate this

[1] The source code is available at rtds.aueb.gr/index.php/software/probslack/
[2] https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/operators/process_function.html

additional constraint in Problem 2.2, rewriting Eq. 7 as follows

$$\mathbb{P}_{\text{miss}}(k,t) = \mathbb{P}[e_{l+1}.gts \leq k \cdot f \wedge t < e_{l+1}.rts \leq t + mal] \quad (8)$$

Dual problem — Some applications may deem more relevant to bound *SL* than *MER* (*e.g.*, applications that are insensitive to missed events). As such we can define the dual of Problem 2.2 as follows.

Problem 3.3 (Window Slack Latency Optimization). *Given a sliding window query* $SWQ = \langle f, l, p \rangle$ *and a user tolerance to slack latency* Δ, *the* Window Slack Latency Optimization *problem aims at finding a rewriting of* SWQ *to* $RSWQ = \langle f, s(k), l, p \rangle$ *s.t.*

$$\min \; MER(RSWQ) \quad (9)$$

$$\textit{subject to } SL(RSWQ) \leq \Delta \quad (10)$$

ProbSlack can be naturally re-framed to fit this scenario as well. As a future work we plan to optimize and evaluate ProbSlack for these two variants.

4 EVALUATION

In this section we present the simulator and prototype we devised for the experiments and introduce a thorough empirical evaluation. We first detail the evaluation setup (Section 4.1) followed by a description of both synthetic and real-world datasets (Section 4.2). The empirical evaluation results for the simulator are presented in Section 4.3, while Section 4.4 discusses the prototype evaluation.

4.1 Setup

All experiments were run on a server equipped with Intel(R) Core(TM) i7-7820X 3.60GHz (8 hyper-threaded CPU cores) and 64 GB RAM. The simulator is a straightforward implementation of the trigger function detailed in Listing 1.

Algorithms — We implemented and tested the following five algorithms:

- BL-ED — Baseline *event-driven* policy where $\text{cond}(k,t,e) = e.gts > k \cdot f$.
- BL-IG — Baseline *ignore* policy where $\text{cond}(k,t,e) = t \geq k \cdot f$.
- BL-AD — A *user-defined* policy with the slack set to the average observed network latency [15], so that $\text{cond}(k,t,e) = t \geq k \cdot f + \overline{d}$.
- ProbSlack — The proposed solution.
- Oracle — Offline optimal algorithm for Problem 2.2. Oracle first computes the *event-driven* $s(k)$ for all windows and then sets to $-f$ the $\delta n/f$ largest $s(k)$.

BL-ED, BL-IG, and BL-AD provide a set of baselines that use common policies to handle late arrivals while Oracle shows a competitive gap to reach an optimal solution. Except the Oracle, the algorithms have been implemented in the cond function.

Metrics — We measure the performances of all algorithms using three metrics, as follows.

- Missed Event Ratio $MER(alg)$ — is the ratio of windows processed by algorithm *alg* with at least one missing event

$$MER(alg) = \frac{f}{n} \sum_{k=1}^{\frac{n}{f}} \mathbb{I}_{k,alg}$$

where the indicator $\mathbb{I}_{k,alg} = 1$ if $E_k - E_k^{alg} \neq \emptyset$ specifies whether *alg* triggered the processing of window *k* with a missing event.

- Average Slack Latency $\overline{SL}(alg)$ — is the average value of the algorithm slack $s_{alg}(k)$ for all windows

$$\overline{SL}(alg) = \frac{f}{n} \sum_{k=1}^{\frac{n}{f}} s_{alg}(k)$$

- Average Completion Time $\overline{CT}(alg)$ — is the average end-to-end latency, computed as the time an event takes to traverse the CEP application.

Notice that, by design, BL-ED has $MER = 0$ and BL-IG has $\overline{SL} = 0$ ($\forall k$, $s_{\text{BL-IG}}(k) = 0$). Moreover, both Oracle and ProbSlack, given sufficient evidence, may close a window before its end ($s_{\text{Oracle}}(k)$,$s_{\text{ProbSlack}}(k) \not> 0$) and their $\overline{SL}$ may be negative.

Parameters — We evaluates the algorithms for several values of the frequency $f \in \{5, 10, 15, 20\ , 25\ , 30, 35, 40\}$ in milliseconds. ProbSlack has two parameters, the *MER* budget δ and the fitting period T. We show a sensitivity analysis with respect to $\delta \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$. Unless specified otherwise, default values are $\delta = 0.1$, $T = 10^4$ and $f = 30$.

4.2 DataSets

The algorithm has been tested against both synthetic and real-world datasets, the former providing the ground for the analysis in a controlled environment and the latter validating the algorithms against a realistic scenario. In particular, there is a lack of publicly and easily accessible event stream datasets with variance in generation and network delay. To overcome this shortage we designed a smart city application and collected the resulting event streams. The dataset repository will be made available in the final version.

Synthetic Datasets — Synthetic streams are made of 10^5 events. Inter-generation and network delays (in milliseconds) are generated with five different combinations of constant, Zipfian (power-law) and Binomial discrete distribution as follows

- CB — inter-generation delay constantly equal to 20 milliseconds and network delay binomially distributed ($p = 0.5$) in $[1, 11]$ with an average of 6 milliseconds.
- BB — inter-generation delay binomially distributed ($p = 0.25$) in $[15, 35]$ with an average of 20 milliseconds and network delay binomially distributed ($p = 0.5$) in $[1, 11]$ with an average of 6 milliseconds.
- BZ — inter-generation delay binomially distributed ($p = 0.25$) in $[15, 35]$ with an average of 20 milliseconds and the network delay distributed Zipf($\alpha = 0.2$) in $[1, 11]$ with an average of 6 milliseconds.
- ZB — inter-generation delay distributed Zipf($\alpha = 1.1$) in $[15, 35]$ with an average of 20 milliseconds and network delay binomially distributed ($p = 0.5$) in $[1, 11]$ with an average of 6 milliseconds.
- ZZ — inter-generation delay distributed Zipf($\alpha = 1.1$) in $[15, 35]$ with an average of 20 milliseconds and network delay distributed Zipf($\alpha = 0.2$) in $[1, 11]$ with an average of 6 milliseconds.

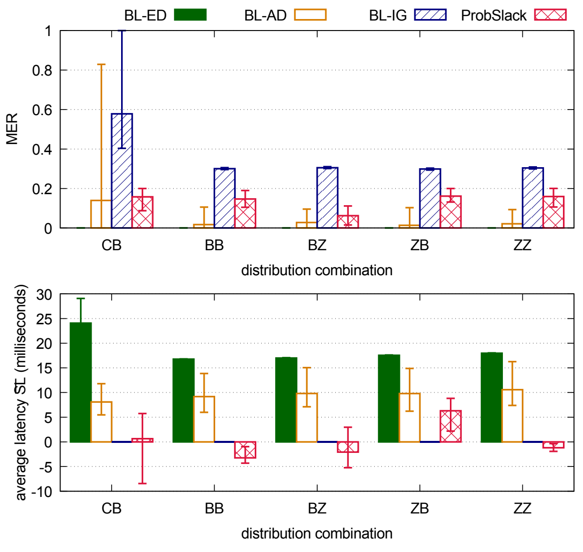

Figure 6: ***MER*** **and** $\overline{SL}$ **as a function of the frequency distribution for all values of** f **and** $\delta = 0.3$**.**

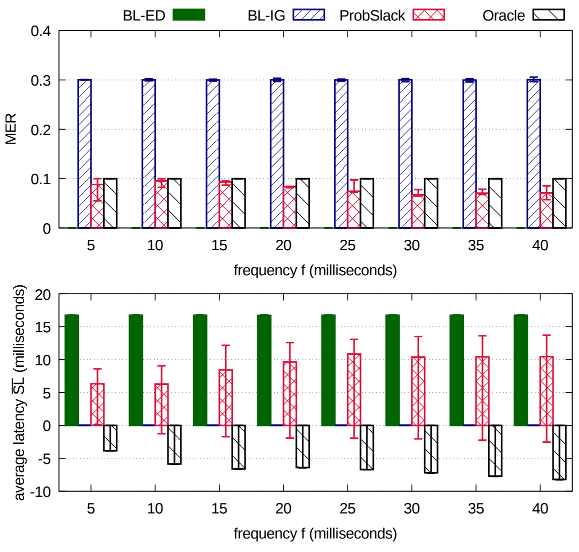

Figure 7: ***MER*** **and** $\overline{SL}$ **as a function of the frequency** f **with** $\delta = 0.10$ **and frequency distribution combination BB.**

Unless specified otherwise, the synthetic stream is generated using the BB combination. We have generated 50 different streams for each combination, using 50 different seeds. We show the average performance over these 50 runs, as well as the minimum and maximum values (as error bars).

Real-world Dataset — We use a common scenario in a smart city application, motivated by a real smart city deployment.[3] Sensors are deployed across Dublin (Ireland) and their output is collected and preprocessed (*e.g.*, compression) in Athens (Greece), before forwarding the events to Haifa (Israel), where the CEP engine is located, for further processing. We are then interested in the timing properties of the preprocessing in Athens, which may exhibit variance in the inter-generation delay, depending on the processing, as well as in the network delay between Athens and Haifa, depending on the taken path and congestion status. More in details, the concentrator collects output stream of sensors mounted on top of 900 public buses and compresses the input stream coalescing the readings that are spatially close. Collecting the stream generation and reception timestamps we built a realistic dataset dubbed SCDS.[4] Figure 1 shows the inter-generation g and network d delays. The minimum, average, and maximum inter-generation delays are 1, 120, and 1557 milliseconds, respectively. The minimum, average, and maximum network delays are 146, 298, and 999 milliseconds, respectively.

4.3 Simulator Results

In this section we present an analysis of experiments that were performed using the simulator.

Sensitivity analysis with respect to g and d — Figure 6 shows MER (top) and $\overline{SL}$ (bottom) for all algorithms, except Oracle, as a function of the synthetic streams g and d probability distributions for all values of $f \in \{5, 10, 15, 20, 25, 30, 35, 40\}$ and $\delta = 0.3$. The large difference between min and max values are due to the aggregation over multiple f values. This plot shows that ProbSlack satisfies the budget $\delta = 0.3$ (MER is at most 20% and on average 13.7%) while introducing a sizable speed-up with respect to BL-ED irrespectively of the frequency distribution combination. In particular, ProbSlack is on average 7.6 times faster than BL-ED and 2.1 times faster than BL-AD. Given sufficient evidence ProbSlack is able to close window k before time $k \cdot f$, then $s_{\text{ProbSlack}}(k) < 0$ and the overall $\overline{SL}$ may be negative. As expected, BL-ED has $MER = 0$, not missing any event, while BL-IG has a high MER value (on average 35%), missing a large number of events. By construction $s_{\text{BG-IG}}(k) = 0$, which yields $\overline{SL} = 0$. For all frequency distributions, BL-AD has non negative values of MER (on average 5%) while being (on average 3.1 times) slower than ProbSlack. While CB seems to be the easiest combination to handle (constant inter-generation delay), both BL-AD and BL-IG have a large MER. This behavior shows that *user-defined* policies do not provide foreseeable performances.

We have evaluated all algorithms in all mentioned parameter configurations and distribution combinations. However, for the sake of brevity, here onward we only report the results for BB as the outcome for the other distribution combinations is comparable. Moreover, BL-AD exhibits similar behaviors across all simulator experiments and for the sake of conciseness, in what follows we shall not report on its performance.

Sensitivity analysis with respect to the frequency f — Figure 7 shows MER (top) and $\overline{SL}$ (bottom) for BL-ED, BL-IG, Oracle and ProbSlack, as a function of the frequency f with $\delta = 0.10$ and frequency distribution combination BB. This plot shows that ProbSlack satisfies the bound $\delta = 0.10$ for all frequencies (on average 8%), while introducing sizable speeds-ups. With respect to BL-ED, ProbSlack is on average 1.9 times faster while being at least 1.2

[3] H2020 VaVeL project www.vavel-project.eu/
[4] The dataset is available at rtds.aueb.gr/index.php/software/probslack/

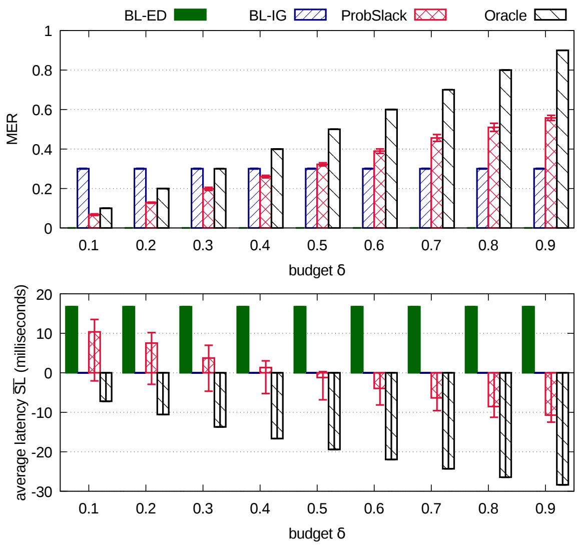

Figure 8: MER **and** $\overline{SL}$ **as a function of the budget** δ **with** $f = 30$ **milliseconds and frequency distribution combination BB.**

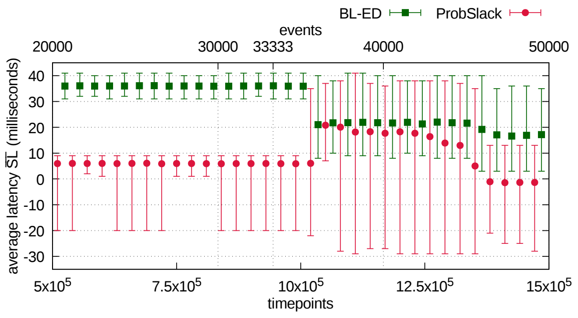

Figure 9: $\overline{SL}$ **time series with** $f = 30$, $\delta = 0.1$ **and with changing frequency distribution combinations.**

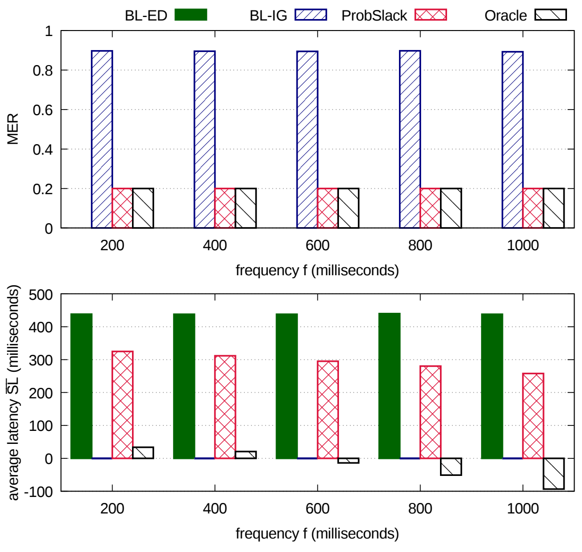

Figure 10: MER **and** $\overline{SL}$ **as a function of the frequency** f **with** $\delta = 0.2$ **and dataset SCDS.**

times faster. It is worth noting that the $\overline{SL}$ gain introduced by ProbSlack with respect to BL-ED decreases for values of f that are large with respect to the average inter-reception time. ProbSlack also outperforms BL-IG, which is 3.7 times less accurate while introducing only a 2 times larger gain with respect to BL-ED $\overline{SL}$. On average, Oracle has a $\overline{SL}$ gain with respect to BL-ED 3 times larger than ProbSlack while having close MER values.

Sensitivity analysis with respect to the budget δ — Figure 8 shows MER (top) and $\overline{SL}$ (bottom), as a function of the budget δ with $f = 30$ and frequency distribution combination BB. This plot shows how, without ever violating the budget δ, ProbSlack provides predictable performances. In particular, MER grows with δ, from 6.7% to 55.7%, while $\overline{SL}$ decreases from 10 to -10 milliseconds. As expected, BL-IG and BL-ED are insensitive to δ, with a respectively constant MER of 30% and 0%. Oracle's MER fits exactly the budget δ, while $\overline{SL}$ decreases from -7.2 to -28.3 milliseconds. On average, Oracle has a $\overline{SL}$ gain with respect to BL-ED 2 times larger than ProbSlack while having close MER values.

Time series with changing frequency distributions — Figure 9 shows the $\overline{SL}$ time series for BL-ED and ProbSlack with $f = 30$, $\delta = 0.1$. Here we restrict the comparison to BL-ED to focus on the behavior of ProbSlack when g and/or d distributions change. The x-axis is time, expressed as timepoints (bottom) and events sequence numbers (top). The stream is generated with changing frequency distribution combinations. In particular, the first 33,333 events are generated with $g = 20$ and $d \sim B(0.5, 1, 11)$ (binomially distributed in $[1, 11]$ with $p = 0.5$), the following 33,333 events are generated with $g \sim B(0.25, 20, 40)$ and $d \sim B(0.5, 6, 15)$, while the final 33,333 events are generated with BZ. It is worth noting that the distribution changes are not synchronized with the fitting period T. To improve the readability of the plot, we provide the average minimum and maximum value of SL over 3000 timepoints, while the plot of BL-ED is shifted on the x-axis. Event 33,333 is received at time 999,992, thus the distributions change start to impact the quality of ProbSlack actions (increasing MER). As expected, ProbSlack detects the change at time 1,004,472 (event 33,483), resets the fitting and switches to the *event driven* policy until the new distribution is learned. Indeed, the $\overline{SL}$ for both ProbSlack and BL-ED is similar from roughly event 36,000 to event 43,500. Once ProbSlack has learned the new distribution at time 1.2×10^6 (event 43,320), the algorithm reverts back to the default behavior. It is noteworthy that ProbSlack has an average MER of 6.7%, never exceeds the budget δ, and is 5.4 times faster then BL-ED.

SCDS datasets results with respect to the frequency f — Figure 10 shows MER (top) and $\overline{SL}$ (bottom), as a function of the frequency f with $\delta = 0.10$ and the SCDC dataset. To align with the SCDS dataset values, we set the frequency values to be $f \in \{200, 400, 600, 800, 1000\}$ milliseconds. The behavior of Oracle is similar to the one presented for synthetic data in Figure 7. On the other hand, the $\overline{SL}$ gain of ProbSlack with respect to BL-ED increases with f. ProbSlack's MER is on average 19% while being 1.49 times faster than BL-ED. We note that the extremely large

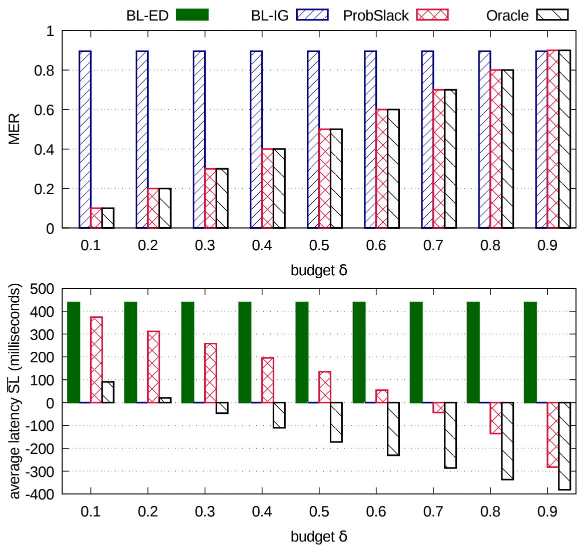

Figure 11: MER **and** $\overline{SL}$ **as a function of the budget** δ **with** $f = 400$ **milliseconds and dataset SCDS.**

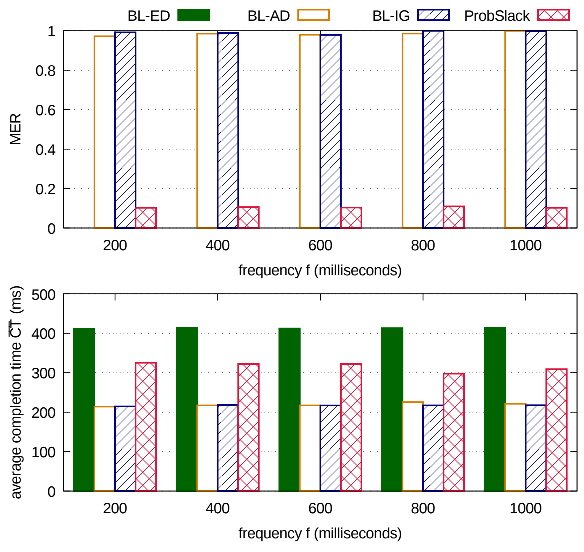

Figure 12: MER **and** $\overline{CT}$ **as a function of the frequency** f **with** $\delta = 0.1$ **and dataset SCDS in the prototype.**

value of $MER > 80\%$, which BL-IG generated, is not counterbalanced by a proportional gain in $\overline{SL}$. On average, Oracle has a $\overline{SL}$ gain with respect to BL-ED 3.1 times larger than ProbSlack while having close MER values.

SCDS datasets results with respect to the budget δ — In Figure 11 we illustrate the MER (top) and $\overline{SL}$ (bottom), as a function of the budget δ with $f = 400$ and the SCDC dataset. The behavior of Oracle and ProbSlack is similar to those presented for synthetic data in Figure 8, except that ProbSlack MER is just below the budget δ. ProbSlack is on average 4.5 times faster than BL-ED. As in Figure 10, BL-IG has poor performances, with $MER > 80\%$. On average, Oracle has an $\overline{SL}$ gain with respect to BL-ED 4.5 times larger than ProbSlack while having close MER values.

To summarize, both real-world and synthetic datasets show similar trends, when comparing Figure 10 and Figure 11. As expected from real-world data, algorithms' performance may be less optimal than with a tightly controlled data.

Overhead evaluation — Section 3.1 discussed the space and time complexities of ProbSlack. Table 1 provides a synopsis of these complexities as well as the empirical resource usage of ProbSlack across all the synthetic streams experimental evaluation. In particular, we show the memory usage as the average and maximum number of entries in the fitting and cache data structures. The update and query times are provided as the average and maximum number of $O(1)$ lookups or updates to these data structures over the whole execution. Table 1 shows the results for three configurations of ProbSlack: without caching, with cached P_{miss} and with lazily cached P_{miss}. Notice that each simulation run entails 2×10^6 timepoints, *i.e.*, the function COND is called 2×10^6 times. Considering the non-cached version, the empirical update time (average and maximum) is 4×10^6 (with 2 updates to the fitting data structures per function COND call). The empirical query time has a roughly 10 fold increase with respect to the empirical update time. Notice that this matches the theoretical complexities since the average number of α distinct values is $\overline{N_\alpha} = 10$. The cached version has roughly a 10 fold improvement in the query time complexity while introducing a negligible cost update time (at most 2,062 additional lookups and updates) and in space (at most 7 additional entries). The lazily cached P_{miss} prevents the caching of values of β that are never observed, further reducing the overhead. Furthermore, caching lazily also spreads the additional lookups and updates over multiple COND function calls.

4.4 Prototype Results

This section compares the performances of the implementation of four algorithms, namely BL-ED, BL-AD, BL-IG and ProbSlack in Apache Flink [17]. BL-ED is implemented setting Flink's execution environment stream time characteristics [5] to *event time* (*i.e.*, generation time). Time progress is provided by the watermarking mechanism, attaching to the source operator an ASSIGNERWITHPUNCTUATEDWATERMARKS returning the generation time of the event as the watermark value. BL-AD is implemented by setting Flink's execution environment stream time characteristics to *system time* (*i.e.*, reception time). Time progress is provided by the watermarking mechanism, attaching an ASSIGNERWITHPERIODICWATERMARKS returning the sum of the average observed network latency and the current system time as the watermark value. BL-IG is implemented by setting Flink's execution environment stream time characteristics to *system time* (*i.e.*, reception time). Time progress is provided by the default mechanism. For all three baselines, the $SWQ = \langle f, l, p \rangle$ query is implemented as a keyed sliding window with the appropriate parameters, *i.e.*, KEYBY($busId$).WINDOW(SLIDINGEVENTTIMEWINDOWS.OF(TIME.MILLISECONDS(10000), TIME.MILLISECONDS(f))). The pattern p searches for congestion.

ProbSlack prototype implementation is detailed in Section 3.2.

[5]https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/event_time.html

Cache Version		without caching	cached P_{miss}	lazily cached P_{miss}
Space Complexity		$O(\max\{N_\alpha, N_\beta\})$	$O(\max\{N_\alpha, N_\beta\})$	$O(\max\{N_\alpha, N_\beta\})$
Update Time Complexity		$O(1)$	$O(1)$	$O(1)$
Query Time Complexity		$O(N_\alpha)$	$O(1)$	$O(1)$
Fitting Memory Usage (entries)	average	27	27	27
	max	42	42	42
Cache Memory Usage (entries)	average	0	3	1.6
	max	0	7	7
Update Time (lookups/updates)	average	4×10^6	$4 \times 10^6 + 1509 + 3$	4×10^6
	max	4×10^6	$4 \times 10^6 + 1955 + 7$	4×10^6
Query Time (lookups/updates)	average	1.5×10^7	2×10^6	$2 \times 10^6 + 765 + 1.6$
	max	3.9×10^7	2×10^6	$2 \times 10^6 + 1024 + 7$

Table 1: ProbSlack theoretical and empirical resource usage.

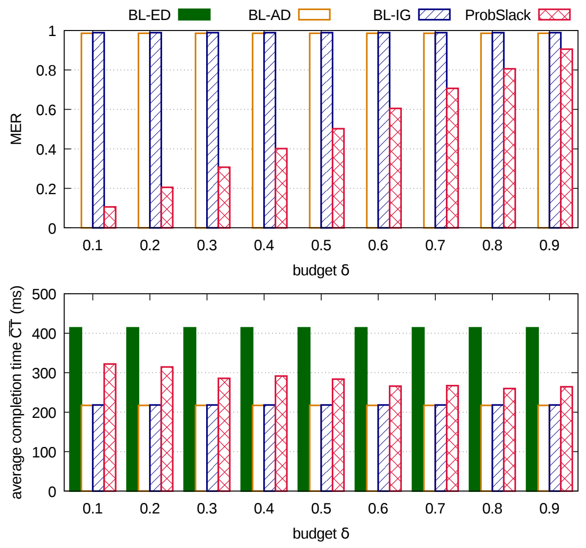

Figure 13: ***MER*** **and** $\overline{CT}$ **as a function of the budget** δ **with** $f = 400$ **and dataset SCDS in the prototype.**

Prototype results with respect to the frequency f — Figure 12 shows *MER* (top) and average completion time $\overline{CT}$ (bottom), for BL-ED, BL-AD, BL-IG, and ProbSlack as a function of the frequency f with $\delta = 0.10$ and the SCDC dataset in the prototype. In both metrics, no algorithm shows a sizable sensitivity to the value of f. As expected, for BL-ED we always get $MER = 0$ coupled with the highest average completion time $\overline{CT}$. Likewise, BL-IG and BL-AD have a high number of misses ($MER > 98\%$) and low completion time ($\overline{CT} < 219$ milliseconds). Unpexpectedly, BL-AD, for which $s(k) = \overline{d}$, has a behavior similar to BL-IG, with a high number of misses ($MER > 97\%$) and a low completion time ($\overline{CT} < 226$). In contrast, ProbSlack provides for all values of f, a low number of misses within by the budget δ, with an average completion latency in between BL-ED and BL-IG (on average 315 milliseconds and at most 325 millisecond). ProbSlack introduces a sizable gain (1.31 times faster) on average completion latency $\overline{CT}$ compared to BL-ED. With respect to BL-IG, ProbSlack is 9.5 times more accurate while being 1.4 times slower.

Prototype results with respect to the budget δ — Figure 13 shows the *MER* (top) and average completion time $\overline{CT}$ (bottom) as a function of the budget δ with $f = 400$ and the SCDC dataset in the prototype. ProbSlack stays just below the budget δ while providing a large gain on average completion time $\overline{CT}$. With respect to the results for the SCDC dataset in the simulator (*cf.*, Figure 11), the average completion time $\overline{CT}$ does not decrease as fast as the $\overline{SL}$ for growing values of δ. On average ProbSlack is 1.46 times faster than BL-ED. Moreover, on average BL-IG and BL-AD are 1.29 times faster than ProbSlack while being 1.95 times less accurate.

Figures 12 and 13 show the gap between simulation and real-world data, while supporting the good performance of ProbSlack and findings of the simulator evaluation.

5 RELATED WORK

One of the most common approaches for handling late arrivals in CEP and stream processing systems are punctuation-based techniques [19]. *Punctuations* are special events in the stream that indicate that no future event is expected with timestamp less than the one carried by the punctuation, *i.e.*, all the events residing on the window can be safely processed. Apache Flink [17] watermarking mechanism is an implementation of the punctuation technique. Most punctuation-based systems assume that the punctuation timestamp is provided by the user or by some external source. However, this is not a realistic assumption for most applications, including the ones we examined in our empirical evaluation. To alleviate this problem, an adaptive punctuation mechanism has been proposed in [15] where the goal is to capture the skew between different streams, the latency and the disorder within streams. With respect to ProbSlack, they do not consider the fact that event inter-generation time may change over time even in the same input stream. Moreover, Srivastava and Widom [15] do not provide guarantees in terms of the amount of missed events (*i.e.*, quality).

Another widely utilized technique for handling late arrivals are buffer-based data structures stalling the processing of incoming events to avoid determinism violations. One of the first stream processing systems Aurora [1], enabled the end-user to set a fixed buffer size. There has been a plethora of previous works [5–8, 21] trying to strike the right balance between accuracy and latency, adjusting dynamically the buffer-size during the processing of incoming events. However, most of the techniques target specific streaming operators (*e.g.,* join operators [6] and aggregations [7]). Our approach is operator agnostic as the *MER* metric (*cf.*, Section 2) does not depend on the operator's semantics. Furthermore, these works take into account statistics only about changes is the overtime network delay [21]. ProbSlack models event inter-generation time and the experienced network delay in probabilistic terms, leading to improved delay estimation.

A third approach for handling late arrivals is using speculation [14]. Tuples are processed without any stalling on the operator's input buffers. In case of late arrivals, the output stream is rolled back and recomputed. Whenever late arrivals are sparse, such techniques can improve significantly the latency of the processing. However, in many applications late arrivals are the norm rather than the exception. Therefore, the constant re-computation of the results affects performance significantly. Mutschler and Philippsen [11] combined the buffer-based and the speculation techniques using the K-Slack data-structure, which enables them to dynamically adapt the amount of time that an event resides in the system before it gets processed. The authors focus on minimizing the observed latency, which impacts the accuracy of the provided results. ProbSlack considers both latency and accuracy. Furthermore, our approach is orthogonal to speculation, *i.e.*, the probabilistic analysis of ProbSlack can be combined with any speculative technique to further improve the timeliness and correct miss-computations.

Finally, handling possible late arrivals was performed by approximating query results [9, 18]. The events' values are summarized in space efficient data structures (*e.g.*, histograms or sketches) and, similarly to the speculative techniques, an approximation of the results is produced in a timely manner [14]. Some recent works [4, 22] adaptively set the window size of stream processing systems, such as Apache Spark [16], aiming at a balance between system's throughput and event end-to-end latency. While these works also provide mechanisms to determine the appropriate time that events should be processed by the operators, they do not consider the impact of their decisions on the results' accuracy.

6 CONCLUSIONS

In this work we presented a probabilistic framework for handling late arriving events in CEP systems and proposed the use of statistical theories to fit the distributions of inter-generation and network delays per event type for predicting whether an event belonging to a window has yet to arrive. We propose a novel algorithm, ProbSlack, aiming the dynamical determination of the amount of time a complex event time window should remain open based on user-defined tolerance levels on the accuracy of the results. A thorough empirical evaluation, including a prototype implementation targeting Apache Flink [17], highlights the benefits of our approach and its superiority compared to state-of-the-art mechanisms for handling late arrivals in CEP systems.

ACKNOWLEDGMENT

This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project and the Horizon2020 688380 VaVeL project.

REFERENCES

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

[2] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD).* ACM, 1093–1104.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 4 (2015), 28–38.

[4] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC).* ACM, 1–13.

[5] Yuanzhen Ji, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven Disorder Handling for Concurrent Windowed Stream Queries with Shared Operators. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS).* ACM, 25–36.

[6] Yuanzhen Ji, J. Sun, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven disorder handling for m-way sliding window stream joins. In *Proceedings of the 32nd International Conference on Data Engineering (ICDE).* IEEE, 493–504.

[7] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD).* ACM, 889–894.

[8] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-driven Processing of Sliding Window Aggregates over Out-of-order Data Streams. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS).* ACM, 68–79.

[9] Chuan-Wen Li, Yu Gu, Ge Yu, and Bonghee Hong. 2011. Aggressive complex event processing with confidence over out-of-order streams. *Journal of Computer Science and Technology* 26, 4 (2011), 685–696.

[10] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order Processing: A New Architecture for High-performance Stream Systems. *Proceedings of the VLDB Endowment* 1, 1 (2008), 274–288.

[11] Christopher Mutschler and Michael Philippsen. 2013. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *Proceedings of the 7th ACM international conference on Distributed Event-Based Systems (DEBS).* ACM, 147–158.

[12] Nicolo Rivetti, Yann Busnel, and Achour Mostéfaoui. 2015. Efficiently Summarizing Data Streams over Sliding Windows. In *Proceedings of the IEEE 14th International Symposium on Network Computing and Applications (NCA).* IEEE, 151–158.

[13] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach.* Pearson Education.

[14] Esther Ryvkina, Anurag S Maskey, Mitch Cherniack, and Stan Zdonik. 2006. Revision processing in a stream processing engine: A high-level design. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE).* IEEE, 141–143.

[15] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS).* ACM, 263–274.

[16] The Apache Software Foundation. Apache Spark. https://spark.apache.org.

[17] The Apache Software Foundation. Apache Flink. https://flink.apache.org/.

[18] Srikanta Tirthapura, Bojian Xu, and Costas Busch. 2006. Sketching Asynchronous Streams over a Sliding Window. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC).* ACM, 82–91.

[19] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[20] Nikos Zacheilas and Vana Kalogeraki. 2016. Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing Autonomic Computing (ICAC).* IEEE, 65–74.

[21] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2017. Maximizing Determinism in Stream Processing Under Latency Constraints. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS).* ACM, 112–123.

[22] Quan Zhang, Yang Song, Ramani R Routray, and Weisong Shi. 2016. Adaptive block and batch sizing for batched stream processing system. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing Autonomic Computing (ICAC).* IEEE, 35–44.