Event Processing with Bounded Latency

Suresh K. Damodaran and Joshua D. Guttman

The MITRE Corporation

Abstract. Latency is an important metric for event detection in Event Processing (EP) systems, particularly for detecting cyber attacks. We introduce the Happened-Before Language (HBL), an EP language with formal semantics with a linear bound for detection complexity. HBL supports features such as sliding windows, stateful analytics, and attribute-value predicate expressions. HBL processing handles each event only once, with no backtracking. Sensitivity analysis confirms that latency changes little regardless of the event stream arrival rate and the number of predicates in an analytic. The design of HBL was motivated by the problem of detecting cyber attacks, especially on cyber-physical systems. HBL is well suited for use by end-points or edge devices due to its linearly bounded detection algorithm.

1 Introduction

The Event Processing (EP) paradigm that combines the sophisticated event pattern specifications of Complex Event Processing (CEP) and the stream processing capabilities of Event Stream Processing (EPS) has evolved over the past decade [9]. These systems are primarily used in enterprise environments with high computing resource availability. As an example, the recent work by Gao et al. [13] demonstrate that the event processing approach yields detection latency below 2 seconds for anomalies arising from cyber attacks in an enterprise environment.

The growth of new IoT devices combined with the legacy Operation Technology (OT) devices and systems with cyber and physical aspects are called *cyber-physical systems* (CPS) or *edge* systems. The potential for cyber attacks on these systems raises the need for event processing techniques that can support the sophisticated features of CEP with low latency detection, and automated response. Such detection may be implemented on event streams generated by sensors in these devices. At the edge, for cyber attack detection, the most important performance metric for an EP system is processing-time latency as defined in [18], because if the attack can be detected quickly enough, valuable response time can be gained.

Achieving low-latency in an EP system is a harder goal when the analytics could include the sequencing operator with negation [34], sliding windows [6,2], stateful analytics where the attributes of a current event is dependent on the existence or non-existence of events previously occurred in the event stream, aggregation operators such as max and average, and event-attribute predicate

expressions [34]. Yet, we aim to achieve exactly this goal of low latency even while processing highly complex and stateful analytics.

The event queries, or analytics, implemented in our detection algorithm are called *watchpoints*. In the rest of the paper, we use the terms analytics and watchpoints interchangeably. Watchpoints are specified in a new domain specific language, called the *Happened-Before Language* (HBL)[8]. The naming of HBL is a nod to Leslie Lamport [19]. In HBL, the *SEQ* operator [34], is termed the Happened-Before (HB) operator. Our detection algorithm supports HB operators that can avoid specific events, supports time and tuple-based sliding windows, stateful analytics, aggregation operators, event attribute predicate expressions for value-based constraints, and recursive analytics. The algorithm achieves linearly bounded latency of detection with sliding windows. Our Java implementation of HBL has a process-time latency of under a millisecond in all cases that we tested.

Since a stream of events can be generated from stored events through replay, the HBL detection algorithm can also be applied equally well to query events previously collected and stored. We also validate the theoretical results using our Java implementation.

Contributions: Our first contribution in this paper is the comprehensive proofs of correctness of existing and novel streaming analytics language features in HBL. For example, standard features such as stateful analytics, as well as novel features such as disjunction, composability, and recursion in analytics are proven to be correctly defined in HBL. Our second contribution is the proof of linear performance bound for process-time latency, as defined by Karimov et. al [18], in the HBL detection algorithm with or without sliding windows. The HBL detection algorithm is also formally shown to be insensitive to the number of HB operators in an analytic, and is shown to process each event once, and only once. An important feature of this algorithm is its ability to process multiple input events concurrently. The scalability of the algorithm is analyzed with respect to concurrency, and is shown to be capable of delivering constant time latency for many types of analytics with adequate support for concurrent processing, allowing HBL detection to trade-off between latency and computational resources. Our third contribution is the experimental proof for the theoretical results presented in this paper.

Structure of this paper. In Section 2, we introduce a few examples from cyber-physical systems and enterprise systems to motivate our design choices for HBL. Section 3 introduces some basic definitions and notations. The abstract syntax and semantics of HBL, specified as a sequence of Single Width Expressions (SWEs) separated by the HB operator, are presented in Section 4. This section also contains the algorithm for event detection. This section does not address the complexity of evaluating a SWE, and that is the subject of Section 5. The sensitivity of the algorithm described in Section 4 is explored in Section 6. A brief survey of related works is provided in Section 7, and Section 8 concludes this paper.

2 Motivating Examples

In this section, we provide three examples from the cyber domain where a low-latency sliding window algorithm for event detection will be useful. The first two examples are from the cyber-physical systems, and the third one is from a Windows™ system. The examples presented below is a subset of the watch-points we have implemented using HBL.

GPS Spoofing Detection: In a GPS Spoofing attack such as [17], incorrect GPS signals deceive receivers, so they report wrong location coordinates. To detect this spoofing, whenever the location coordinates change faster than a threshold, an alert can be generated. The threshold is set based on the physical limits of the system. One must temporarily store recent time and location information as new GPS data arrives, and evaluate the differences in location within a fixed duration, and flag an alert when the differences in location are beyond expected limits. While this evaluation must be conducted using a sliding window, starting with every new record with a GPS location data using its time stamp, the time window itself can be adjusted to be fairly small to detect such a spoofing attack. Bounded detection latency is important in this case, because without it, the vehicle that uses the GPS navigation system could be at the wrong place at the wrong time with unpleasant consequences.

Infusion Pump Attack Detection: An infusion pump is directly connected to the patient and discharges the prescribed amount of a medicine such as insulin into the patient's blood stream. An attack on the pump [23] that causes the infusion pump syringe to move beyond the prescribed limits for that medicine will need to be detected to prevent an adverse reaction on the patient. The movements of the syringe can be detected using a distance sensor. The violation of the limits can be detected using a sliding window approach on the sensor measurements. Since the future positions of the syringe are dependent on the current position of the syringe, the limits are relative to the current position. Therefore, similar to the GPS spoofing attack described above, the infusion pump attack also does not require a large sliding window. Low-latency is clearly important in this case for the well-being of the patient.

Process Spawning Chain Detection: PowerShell is a scripting environment included within Windows that is used by both attackers and administrators. Execution of PowerShell scripts in most Windows versions is hidden from users and not typically secured by antivirus programs, which makes using PowerShell an easy way to circumvent security measures. A trojan attack reported by Carbon-Black [21] reports an attack sequence of spawning PowerShell from `Outlook.exe`. The attack starts by spawning `winword.exe` when a user clicks on a malicious file running a macro that spawns `cmd.exe`, which in turn spawns `powershell.exe`. Detecting this attack requires tracking all the `Outlook.exe` processes and their children, eventually leading to the `powershell.exe`. The detection algorithm needs to maintain the states of spawning, and report an alert when PowerShell

is spawned through a process chain from `Outlook.exe`. In this case, the sliding-window size is not fixed, and is not within the control of the analytic. However, if the `Outlook.exe` process exits, all the states and information related to that `Outlook.exe` process can be discarded.

3 Preliminaries

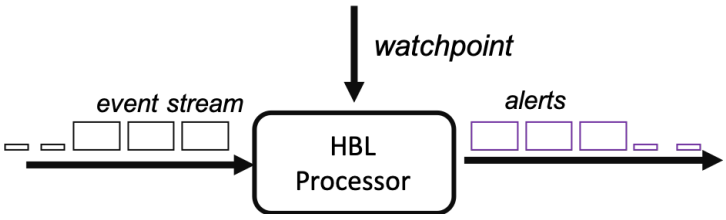


Fig. 1: Detection Context

An HBL watchpoint expression, or watchpoint, specifies a complex event pattern to be detected within a stream of events, and generates one or more alerts corresponding to one or more event sequences that match the pattern (see Section 4 for a formal definition). An HBL Processor implements the watchpoint matching algorithm, and takes a number of *watchpoints*—specifications of the event patterns it should report. An HBL Processor then processes a stream of incoming *events*, in response to which it emits a stream of *alerts*, one for each time an input event segment completes a match against a watchpoint (Fig. 1).

Event Stream: A finite or infinite sequence of events is an *input event stream* [2]. An event is a *record*, i.e. a tuple with fields such as an event type, time stamp, sender, location, or IP address. The structure of records is largely irrelevant, as is how successive records are distinguished in its input stream.

Notation. REC is the set of possible records. A possible sequence of records a *trace*. Formally, a trace is a function from numerical indices to records $r \in \text{REC}$. The domain of a trace is an $\mathsf{Interval}$, either all the natural numbers $\mathbb{N}$ or an initial segment of $\mathbb{N}$:

$$\mathsf{Seg}_i = \{j : 0 \leq j < i\} \tag{1}$$

$$\mathsf{Interval} = \{\mathbb{N}\} \cup \{\,\mathsf{Seg}_i : i \in \mathbb{N}\} \tag{2}$$

$$\text{TR} \subseteq \{\tau : I \to \text{REC} \text{ where } I \in \mathsf{Interval}\}. \tag{3}$$

We use τ to range over traces. Not every function from an interval to records is a trace. For instance, some input streams have record numbers that increase monotonically. Others may have non-decreasing timestamps, or timestamps that are always within some bound of the largest seen value.

If $\tau \in \text{TR}$ is a trace and $i, j \in \mathbb{N}$, then:

$D_1 \times D_2$**:** the domain of pairs of elements of D_1 and D_2.
$D_1 \to D_2$**:** the domain of functions from D_1 to D_2.
$D_1 + D_2$**:** the set of tagged values $\{(\mathsf{left}, d_1) \colon d_1 \in D_1\} \cup \{(\mathsf{right}, d_2) \colon d_2 \in D_2\}$.
When $D_1 \cap D_2 = \emptyset$, we tacitly omit the tag.
$D_\perp$: lifts D by augmenting it with a *bottom* element $\perp$.
For functions with range $D_\perp$, $\perp$ represents non-termination. $\perp$ also represents undefinedness, e.g. $\tau \mathbin{@} i$ for too large an i. All function symbols f we define are *strict*, i.e. $f(\perp) = \perp$.

Table 1: Domain operators applied to domains D, D_1, D_2

$\tau \mathbin{@} i$ means $\tau(i)$, the record at position i in τ.
$|\tau|$ is the length of τ, when $dom(\tau)$ is finite, i.e. $\tau \colon [0, j) \to \text{REC}$.
$\perp$ means undefinedness. E.g, $\tau \mathbin{@} i = \perp$ in case $|\tau| \leq i$.
$\tau[i, j]$ is the *stretch* of τ from position i to position j inclusive. If $j < i$ or $|\tau| < j$, then $\tau[i, j]$ cannot be well-defined. Formally, $\tau[i, j] = (\tau, i, j)$, but we use $\tau[i, j]$ for the sequence of records $\langle \tau \mathbin{@} i, \ldots, \tau \mathbin{@} j \rangle$, e.g. if $\tau[i, j]$ is a segment of τ matching e.
$\tau \dagger i$ is the part of τ following the first i records. Formally, $(\tau \dagger i) @ j = \tau @ (i+j)$.
The empty list is $\langle\rangle$. The list with head a and tail ℓ is $a :: \ell$.

We consider REC, TR, and $\mathbb{N}$ as data types, or "domains," that are used by the HBL Processor. Table 1 gives operators used later in the semantics of HBL.

Environment. A watchpoint is processed by the HBL Processor within the context of an *environment*. An environment may have variables. The HBL Processor binds values derived from the fields of records to these variables. The HBL Processor may consult these variables when evaluating a newly arrived record to match a watchpoint. These variables thus provide a mechanism for storing information from previously encountered records for use in processing future records.

Formally, the domain ENV of environments is the set of partial functions from variables to values. Variables η, η', η_1, etc. range over environments. The next two sections describe the semantics of HBL in more detail.

4 The Happened-Before Language (HBL)

In this section, we define the syntax and semantics for the Happened-Before Language (HBL). The syntax shown in this paper is an abstracted version of the syntax used in our Java implementation of HBL.

We use a, a_1, b, etc. to refer to single-width expressions in SWE and e, e_1, e', etc. for HBL watchpoint expressions. We also refer to these expressions below simply as watchpoints.

Watchpoint Categories: The watchpoints are of three kinds. First, there are *single-width expressions* (SWEs). A single-width expression a is satisfied or

matched by a single event record. A single-width expression can contain a negation [34], e.g. $!a$, where $!a$ is satisfied by any single record that does not satisfy a. It may also use disjunction $a \vee b$, conjunction $a \wedge b$, and so on. However, a match is always a single record.

Second, a watchpoint $a \rightarrow e$ matches a segment of the input. It must begin with a record satisfying the single-width expression a, followed (not necessarily immediately) by segment matching the remainder e. Since a matches a single record and e matches a segment of length one or more, $a \rightarrow e$ matches only segments of length ≥ 2.

Third, a watchpoint $a \rightsquigarrow \underline{b} \rightsquigarrow e$ matches a segment starting with a record satisfying a, and is followed by segment matching the remainder e; however, it must also *avoid* any record satisfying the single-width expression b after the record satisfying a and until the match to e begins. A match to $a \rightsquigarrow \underline{b} \rightsquigarrow e$ is also of length ≥ 2.

Avoid expressions are *not* negations. In a match to the avoid expression $a \rightsquigarrow \underline{b} \rightsquigarrow e$, there may be no record that matches $!b$. The match to e may start immediately after the record that satisfies a, so the avoidance requirement is satisfied "vacuously."

To keep these considerations clear, we divide our description of the Happened-Before Language into two sections. First, in this section, we describe the watchpoints, by assuming there is a *satisfies* relation that determines whether a single record succeeds in matching a single-width expression or fails. Subsequently, in Section 5, we will the *satisfies* relation of SWE.

4.1 HBL Syntax

Let us define the different types of watchpoints that we discussed in the previous section more formally now.

Definition 1. *An* HBL *watchpoint may consist of:*

- a**,** *an* SWE*, meaning that the* HBL *Processor will report the single width stretch* $\tau[i,i]$ *whenever the record* τ @ i *satisfies* a*.*
- $a \rightarrow e$**,** *meaning that the* HBL *Processor will report the stretch* $\tau[i,j]$ *whenever* τ @ i *satisfies* a*; the sub-stretch* $\tau[k,j]$ *offers a match to* e*; and no partial match to* e *starts in* $\tau[i+1,k]$*, where* $i < k \leq j$*. This last condition expresses the eager (or greedy) approach to looking for a match to* e*. We call* $a \rightarrow e$ *an* arrow *expression.*
- $a \rightsquigarrow \underline{b} \rightsquigarrow e$ *also requires* avoiding *satisfying the* SWE b *between* a *and the start of* e*. Specifically, it means that* $a \longrightarrow e$ *holds, and, moreover, in the stretch* $\tau[i+1,k]$ *between the record satisfying* a *and the beginning of the following match to* e*, there should be no record satisfying* b*. We call* $a \rightsquigarrow \underline{b} \rightsquigarrow e$ *an* avoid *expression, and say that* $\underline{b}$ *is its* avoided condition.

The lead *of an* HBL *expression* e*, written* $ld(e)$*, where* e *has one of the forms* a*,* $a \rightarrow e$*, or* $a \rightsquigarrow \underline{b} \rightsquigarrow e$*, is the* SWE a*.*

The importance of $ld(e)$ is that any match to e must begin with a record satisfying a.

The *avoid expression* $a \rightsquigarrow \underline{b} \rightsquigarrow e$ is a single operator with three parameters. It is not built from parts of the form $a \to b$, $b \to e$ or $c \to e$, where c is satisfied by records that do not satisfy b, etc.

Observe that an HBL expression never begins or ends with an underlined SWE $\underline{b}$. Thus the syntax can never yield two successive underlined SWEs $\ldots \underline{b} \rightsquigarrow \underline{c} \ldots$. There is always a non-avoided SWE in between, such as c in $a \rightsquigarrow \underline{b} \rightsquigarrow c \rightsquigarrow \underline{d} \rightsquigarrow e$.

In $a \to e$ and $a \rightsquigarrow \underline{b} \rightsquigarrow e$, the HBL Processor looks for a greedy match to e; i.e. it starts an attempted match at the first record that could start a match to e, after which it will not backtrack to look again at previous records. Therefore, HBL Processor will evaluate an event record once, and only once in the processing of a watchpoint.

We will next give a precise semantics for HBL when environments are in use.

4.2 HBL Semantics

Since HBL watchpoints aim to correlate event records with value-based constraints, they process the stream statefully, using *environments* that the HBL processor maintains.

The goal of the HBL Processor, when given a watchpoint e and an input stream τ, is to detect the *stretches* $\tau[i, j]$ that provide matches to e, starting off from some initial environment $\eta_0 \in \text{ENV}$. Thus, the overall goal of the semantics is to predict what the right answer to this question should be, as a function of e, τ, and the choice of η_0. As a practical matter, the HBL Processor also stores various useful pieces of information into the successive environments as it computes, such as the indices of various partial matches within τ. Thus, we in fact want the pairs $(\tau[i, j], \eta)$ such that matching e succeeds from i to j and results in the final environment η.

The form of the semantics. To give the semantics in a compositional way, we define $[\![e]\!]$ to take as arguments the environment η in force at any point in the matching, the trace, and index i for the start of a match. The result for $[\![e]\!]_\eta \tau i$ is a pair j, η' such that $\tau[i, j]$ is a shortest match starting at i, and η' is the environment resulting at the end of the matching. Alternatively, if i is not the start of a successful match, $[\![e]\!]_\eta \tau i = \bot$. Thus:

$$[\![e]\!] : \text{ENV} \to \text{TR} \to \mathbb{N} \to (\mathbb{N} \times \text{ENV})_\bot \tag{4}$$

We also regard $[\![e]\!]_\eta$ as determining a set of stretches:

$$\tau[i, j] \in [\![e]\!]_\eta \quad \text{iff} \quad \exists \eta' \,.\, [\![e]\!]_\eta \, \tau \, i = j, \eta'. \tag{5}$$

$\tau[i, j] \in [\![e]\!]_\eta$ always implies that $i \leq j$. That is, we are interested only in stretches of τ that include at least one record.

The semantics of HBL is parameterized by a choice of syntax and semantics for the single-width expressions SWE. The results of this section are essentially

$$\frac{\textsc{sat}_\eta(a,\tau,i) = \eta'}{[\![a]\!]_\eta\,\tau\,i = i,\eta'}$$

$$\frac{\begin{array}{c}\textsc{sat}_\eta(a,\tau,i) = \eta_1 \qquad [\![e]\!]_{\eta_1}\,\tau\,k = j,\eta' \qquad i < k \le j \\ \mathsf{fails}_{\eta_1}(ld(e),\tau,i+1,k-1)\end{array}}{[\![a \to e]\!]_\eta\,\tau\,i = j,\eta'}$$

$$\frac{\begin{array}{ccc} & \textsc{sat}_\eta(a,\tau,i) = \eta_1 \qquad [\![e]\!]_{\eta_1}\,\tau\,k = j,\eta' & \\ i < k \le j \qquad \mathsf{fails}_{\eta_1}(ld(e),\tau,i+1,k-1) & & \mathsf{fails}_{\eta_1}(b,\tau,i+1,k-1)\end{array}}{[\![a \rightsquigarrow \underline{b} \rightsquigarrow e]\!]_\eta\,\tau\,i = j,\eta'}$$

Fig. 2: Semantic rules for HBL

independent of the choice of the SWE language. We assume only that the SWE semantics determines a *satisfaction* function:

$$\textsc{sat}(a,\tau,i)\colon \textsc{env} \to \textsc{env} + (\{\mathsf{False}\} + \{\mathsf{Never}\}). \tag{6}$$

SAT decides, for any $a\colon$ SWE, and any record $r = \tau @ i\colon$ REC, whether r satisfies a under environment η, which we generally write $\textsc{sat}_\eta(a,\tau,j)$. If r does satisfy a, SAT returns an updated environment η'. To report non-satisfaction, SAT returns False.

The result Never reflects the possible monotonicity constraints on τ. When $\textsc{sat}_\eta(a,\tau,i)$ yields a tagged value Never, r does not satisfy a under η, and moreover there can be no $j > i$ such that $\textsc{sat}_\eta(a,\tau,j)$. For instance, once the index i has passed some bound, no later entry j will ever decrease it below the bound. Since False and Never are in themselves distinguishable from each other and from any environment, we have ignored the tags left, right that are present in the definition of $D_1 + D_2$ in Fig. 1.

We say $\textsc{sat}_\eta(a,\tau,i)$ *fails* iff $\textsc{sat}_\eta(a,\tau,i) \notin \textsc{env}$, i.e. $\textsc{sat}_\eta(a,\tau,i) = \mathsf{False}$ or $\textsc{sat}_\eta(a,\tau,i) = \mathsf{Never}$. By $\mathsf{fails}_\eta(a,\tau,i,j)$, we mean that $\textsc{sat}_\eta(a,\tau,k)$ fails throughout the interval $i \le k \le j$.

Semantic rules for hbl The semantics of HBL watchpoints is given by the three rules in Fig 2. The first rule says that, for the stretch $\tau[i,i]$ to be a stretch of the input in which a matches, it suffices that $\textsc{sat}_\eta(a,\tau,i) = \eta'$, after which further matching should proceed from η'.

The second rules says that a stretch $\tau[i,j]$ that matches $a \to e$ consists of a match to a at $\tau[i,i]$, followed by a match to e at stretch $\tau[k,j]$, subject to the requirement that the lead $ld(e)$ fails throughout the intervening records. The ηs are threaded through monadically; i.e. there is an intermediate η_1 generated by the match to a at $\tau[i,i]$ and consumed by the match to e at $\tau[k,j]$, which yields the final η' from which any further matching will proceed.

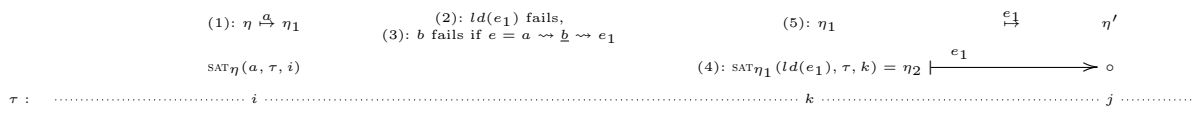


Fig. 3: Conclusions (1)–(5) of Lemma 1

The third rule checks that the avoid SWE b fails for the intervening records. The rules in Fig. 2 express an inductive definition, so that $[\![\cdot]\!]$ is the least fixed point of these rules; this minimality property justifies inductive proof [10].

4.3 Consequences of the semantics

We can now give routine proofs of various consequences of the semantics, using the inductive proof method.

Lemma 1 *If $[\![e]\!]_\eta\, \tau\, i = j, \eta'$, then $j \geq i$. If $e = a$, then $i = j$.*
Otherwise $e = a \to e_1$ or $e = a \rightsquigarrow \underline{b} \rightsquigarrow e_1$, $j > i$ and $\exists \eta_1, k > i$:

1. $[\![a]\!]_\eta\, \tau\, i = i, \eta_1$;
2. $\mathsf{fails}_{\eta_1}(ld(e_1), \tau, i+1, k-1)$;
3. $\mathsf{fails}_{\eta_1}(b, \tau, i+1, k-1)$ *in case* $e = a \rightsquigarrow \underline{b} \rightsquigarrow e_1$;
4. $\exists \eta_2 \,.\, [\![ld(e_1)]\!]_{\eta_1}\, \tau\, k = k, \eta_2$;
5. $[\![e_1]\!]_{\eta_1}\, \tau\, k = j, \eta'$.

Proof. Consider for the successive clauses:

1. The rules for arrow expressions and avoid expressions require the premise $\text{SAT}_\eta(a, \tau, i) = \eta'$, which suffices for $[\![a]\!]_\eta\, \tau\, i = i, \eta_1$.
2. The rules for arrow expressions and avoid expressions require the premise $\mathsf{fails}_{\eta_1}(ld(e), \tau, i+1, k-1)$.
3. The rule for avoid expressions requires the premise $\mathsf{fails}_{\eta_1}(b, \tau, i+1, k-1)$.
4. Applying claim (1) recursively to the subexpression e_1.
5. The rules for arrow expressions and avoid expressions require the premise $[\![e]\!]_{\eta_1}\, \tau\, k = j, \eta'$.

Fig. 3 shows a diagram of these conclusions. We can also define an *composition* function on HBL expressions:

Definition 2. *For any e_1, e_2: HBL, the result of composing e_1 and e_2, written $e_1 \frown e_2$ is defined by recursion on the structure of e_1;*

$$e_1 \frown e_2 = \begin{cases} a \to e_2 & \text{if } e_1 = a \\ a \to (e_3 \frown e_2) & \text{if } e_1 = a \to e_3 \\ a \rightsquigarrow \underline{b} \rightsquigarrow (e_3 \frown e_2) & \text{if } e_1 = a \rightsquigarrow \underline{b} \rightsquigarrow e_3 \end{cases}$$

The semantics of $e_1 \frown e_2$ follows from the semantics of e_1 and e_2:

Lemma 2 $[\![e_1{}^\frown e_2]\!]_\eta\,\tau\,i = j, \eta'$ *iff* $\exists \eta_2, k, \ell$ *such that* $i \le k < \ell \le j$ *and*

1. $[\![e_1]\!]_\eta\,\tau\,i = k, \eta_2$*;*
2. $[\![e_2]\!]_{\eta_2}\,\tau\,\ell = j, \eta'$*; and*
3. $\mathsf{fails}_{\eta_2}(ld(e_2), \tau, k+1, \ell-1)$.

Proof. By induction on the structure of e_1. If $e_1 = a$, then $[\![e_1]\!]_\eta\,\tau\,i = k, \eta_2$ iff $k = i$ and $\textsc{sat}_\eta(a, \tau, i) = \eta_2$. Since $a{}^\frown e_2 = a \to e_2$, the semantics for the latter ensures the claim.

If $e_1 = a \to e_0$, then $e_1{}^\frown e_2 = a \to e_0{}^\frown e_2$. Suppose inductively that the claim holds for e_0 and e_2. Thus again the semantics for $a \to e$ ensures the claim. The case for $e_1 = a \rightsquigarrow \underline{b} \rightsquigarrow e_0$ is similar. □

Lemma 2 says we can faithfully evaluate $e_1{}^\frown e_2$ by evaluating e_1 and e_2 in sequence, starting e_2 with the intermediate environment η_2, and checking that $ld(e_2)$ fails from the completion of e_1 until the selected start index ℓ for e_2.

Lemma 2 suggests our execution strategy. We say that e_1 is an e_0-*derivative* of e iff $e = e_0{}^\frown e_1$. Thus, Lemma 2 says that to find matches to e, it suffices—having found a *partial match* to e_0 and obtaining the environment η_1—to look for a match to the derivative e_1 after an interval in which no records satisfying $ld(e_1)$ are found.

In the next section, we show that the HBL semantics justifies a processing model that maintains a set of *active partial matches* as parts of a watchpoint are matched. Each active partial match moves forward when a record is received that satisfies the watchpoint requirements. The partial match is discarded if a record is received that violates an *avoid* condition. In no case does the HBL processor backtrack and revisit previously seen records.

4.4 Execution model

The HBL Processor maintains a set of *active partial matches* for an expression e it is processing. Each active partial match is a triple, consisting of:

- e_1, the derivative of e that remains to be matched;
- either b, the SWE to avoid from the enclosing expression, or $\perp$ if none; and
- η_1, the environment returned by the partial matching so far.

Execution generates a stream of matches, each represented by a final environment η, as each partial match reaches the empty derivative.

Execution begins—given a set of user-supplied initial watchpoints E_0—with the set of active partial matches $\{(e_0, \perp, \eta_0) \colon e_0 \in E_0\}$, where η_0 is a suitable *initial environment.*

On receiving a new record, with index one greater, we try to "step forward" each active partial match. Each active partial match $\alpha = (e, mb, \eta)$ consists of a derivative e of some initial expression $e_0 \in E_0$, possibly an avoid SWE, and the environment η that resulted from the match encountered so far. For the new record $r = \tau \mathbin{@} i$:

1. For each watchpoint $e_0 \in E_0$, we allocate a new active partial match $\alpha = (e_0, \bot, \eta_0)$, where η_0 is the initial environment.
2. Now, for each active partial match $\alpha = (e, mb, \eta)$, we let $a = ld(e)$ and evaluate $\text{SAT}_\eta(a, \tau, i)$, obtaining one of:
   (a) η' when $r = \tau @ i$ satisfies a. Depending on the form of e:
      i. If $e = a$, we report η' as a new successful match, discarding the now completed active partial match α.
      ii. If $e = a \to e'$, replace α with active partial match $\alpha = (e', \bot, \eta')$.
      iii. If $e = a \rightsquigarrow \underline{b} \rightsquigarrow e'$, replace α with active partial match $\alpha = (e', b, \eta')$.
   (b) Never, so no match will ever occur: we discard α.
   (c) False. If $mb = b$ must be avoided, check $\text{SAT}_\eta(b, \tau, i)$; if it is satisfied, we discard α. Otherwise, we retain α.

This algorithm, described more precisely in Fig. 4 using OCaml [20], contains a procedure `process_one` to process a single record r with a single active partial match a at position i in the stream. Data type definitions and a few auxiliary procedures are shown in Fig. 9, p. 25. The procedure `process_records` calls `process_one` repeatedly, using `io.report` to report the environments of completed, successful matches. It handles a list of watchpoints on which to attempt matches, and it accumulates active partial matches `apm`, `apm'` as they are returned by calls to `process_one`. In this version, we identify each SWE a with the function $\Phi_a(r, i, \eta)$ such that $\Phi_a((\tau @ i), i, \eta) = \text{SAT}_\eta(a, \tau, i)$.

Example 1. Suppose that we have the three watchpoints

$$a \to b \to c, \quad d \rightsquigarrow \underline{b} \rightsquigarrow c, \quad b \to d,$$

which we will refer to as e_1, e_2, e_3 respectively. In this example, the environments do not change; all environments equal the initial environment η_0.

After processing records including an a and then a d, we have copies of the initial `apm`s for e_1, e_2, e_3, plus `apm`s $\alpha_1 = (b \to c, \bot, \eta_0)$ and $\alpha_2 = (c, b, \eta_0)$. The `apm` α_1 is looking for a b followed by a c, while α_2 is looking for a c but will fail if a b is encountered.

If b comes next, α_1 steps to $(c, \bot, \eta_0)$, and α_2 is discarded. Meanwhile, $(e_3, \bot, \eta_0)$ also consumes b and steps to $(d, \bot, \eta_0)$.

Observe that processing any record r with any α requires either one or, if there is an avoid expression, two evaluations of $\text{SAT}(\cdot, \tau, i)$. Thus, it is linear in the maximum cost of evaluating any SWE appearing in the initial, user-specified expression e_0.

Theorem 3 *Suppose that the algorithm of Fig. 4 is executed starting with a (fixed) set S of k watchpoints, and within those watchpoints the worst case time ℓ is needed to evaluate any single* SWE *on any record r at any stream position i.*

1. *Any call to* `process_one` *with a record r and active partial match a takes at most $2 \cdot \ell + c$ time, where c characterizes the auxiliary functions used;*
2. *After processing i records in the stream, there are at most $i \cdot k$ active partial matches;*

```
 let process_one r i a =
match (lead a.e) r i a.env with
  | E eta             ->              (* satisfied the lead *)
    (match remainder a.e with         (* Done or continue? *)
      | None          -> Success eta
      | Some e'       -> Next (make_apm e' a.e eta))
  | Never             -> Fail         (* Will discard apm *)
  | False             ->
    (match a.avoider with             (* Avoid expr? *)
      | None          -> Next a       (* No:  Continue *)
      | Some av       ->
        (match av r i a.env with
          | E _       -> Fail         (* Fail if satisfied *)
          | _         -> Next a))     (* Else, continue *)

let rec process_records : (exp list -> io_struct -> apm list -> unit) =
  fun watchpoints io apms ->
    process_records watchpoints io      (* Call recursively after *)
      (let r = io.get () in             (*   handling next record *)
       let i = io.index () in           (*   at next stream pos *)
       List.fold_left                   (* Collect new apms *)
         (fun so_far apm ->
           match process_one r i apm with               (* apply apm to r *)
             | Fail             -> so_far
             | Next apm'        -> apm' :: so_far       (* apm' is new *)
             | Success eta'     -> (io.report eta';     (* report success *)
                                    so_far))
         []
         (prepend_apms i apms           (* Add to apms:  New starts *)
            watchpoints))               (* for given watchpoints *)
```

Fig. 4: `process_one` delivers a record r to an APM a; `process_records` calls it iteratively.

3. *For fixed k, the time needed for* `process_records` *to handle the i^{th} record is at most $\mathcal{O}(i \cdot \ell)$.*

Proof. Joshua - TBD

The last clause (3) implies that, for a given set of watchpoints (and hence a fixed ℓ) the latency can at worst grow linearly in the length of the stream processed so far.

Example 2. A worst case in clause (2), arises from a single watchpoint $e = a\ \mathsf{letting}(\mathsf{x} = \mathsf{str\_ind}()) \to b$ that looks for an occurrence of a, binds the variable x to the stream index at which it occurs in the environment, and then looks for b.

Executing e against an input stream $\langle a, a, a, a, \ldots, a, b\rangle$, a causes a new active partial match seeking b. These are all distinct, since each one binds x to different index i in the environment.

In Example 2, `apm`s persist, and e triggers a distinct `apm` from each input record. However, in many situations the likelihood of matching on the initial record is low; then new active partial matches will rarely accumulate. When the watchpoints have *avoid* expressions, and the likelihood of a record satisfying the avoid clause is substantial, the pool of active partial matches will again remain modest.

4.5 Regex Processing

As described so far, the HBL watchpoints are a simple subset h of the regular expressions over an alphabet that consists not of individual characters but of all possible records. These regular expressions, REs, are of the three forms:

$$\begin{aligned} h \subseteq \text{RE} ::= {} & a \\ \mid {} & a([\hat{}\ ld(h)]^*)h \\ \mid {} & a([\hat{}\ b,\ ld(h)]^*)h \end{aligned} \tag{7}$$

That is, on the middle line of Equation 7, a match consists of a record satisfying a; a stretch in which these is no match to the lead of the remainder; and a stretch that consists of a match to the remainder h. On the third line, records satisfying the *avoided condition* b must also not be found. This exclusion of partial matches to the lead $ld(h)$ accounts for the simple, predictable execution model, with no backtracking, that HBL Processor executes. Indeed, when the HBL processor knows that part of a watchpoint does not affect the environment, and that the record stream is represented as a string, it uses regular expressions as in Eq. (7) to speed up its processing. Woods et. al [33] use a related trick.

4.6 Sliding Window

The bound in (2) is pessimistic in another way also. In many cases, we have a sliding window w for matching that ensures that any matcher created at step i will be discarded with a Never by step $i + w$. Given such a sliding window, the number of active partial matches is bounded by $w \cdot k$ independent of i, and the bound in (3) is simply $\mathcal{O}(\ell)$.

To define the sliding window functionality carefully, we will say recursively for any watchpoint e: the 0^{th} derivative of e is e; and if e_n is the n^{th} derivative of e, then:

- the $n + 1^{\text{st}}$ *lead* in e is $ld(e_n)$;
- the $n + 1^{\text{st}}$ *derivative* of e is e_{n+1} if either $e_n = a_n \to e_{n+1}$ or $e_n = a_{n+1} \rightsquigarrow \underline{b} \rightsquigarrow e_{n+1}$.
  Otherwise, the $n + 1^{\text{st}}$ derivative is undefined.

$\langle (i_k, \eta_k) \rangle_{0 \le k \le j}$ is *generated by* e *from* stream τ iff, recursively,

- $j = 0$; $i_0 = 0$; and η_0 is the initial environment; or
- $\langle (i_k, \eta_k) \rangle_{0 \le k \le j-1}$ is generated by e from τ and:
  - the j^{th} lead of e is a_j;
  - $\mathsf{fails}_{\eta_{j-1}}(a_j, \tau, i_{j-1}, i_j - 1)$; and
  - $\text{SAT}_{\eta_{j-1}}(a_j, \tau, i_j) = \eta_j$.

Intuitively, the i_k are the positions of the records that satisfied SWEs in e, and η_k are the resulting environments, for $k > 0$.

Definition 3. *A watchpoint* e enforce sliding window w *iff, for any stream* τ*, there is a maximal* $\langle (i_k, \eta_k) \rangle_{0 \le k \le j}$ *generated by* e*, where* $i_k < i_1 + w$ *and the* $j+1^{\text{st}}$ *lead in* e *is either (i) undefined, or else (ii)* a_{j+1} *and for some* $\ell < i_1 + w$*,* $\text{SAT}_{\eta_j}(a_{j+1}, \tau, \ell) = \mathsf{Never}$.

Here, case (i) arises when the earliest match for e in τ succeeds before the sliding window expires, and case (ii) occurs when the window expires with the $j + 1^{\text{st}}$ lead reporting Never.

One may implement windows using stream indices (as in Example 2) and a conditional in a_2. Given a bound on the rate at which new records will arrive in the stream, one can also implement a window using timestamps contained in the records.

Lemma 4 *Suppose that the algorithm of Fig. 4 is executed starting with a (fixed) set* S *of* k *watchpoints, and within those watchpoints at most time* ℓ *is needed to evaluate any single* SWE*, regardless of the record* r *and stream position* i*. Suppose moreover that* S *is chosen so that each watchpoint* $e_j \in S$ *enforces a window* w_j*, with each* $w_j < w$*.*

Then the worst case time for `process_records` *to handle a record is* $\mathcal{O}(w \cdot \ell)$*.*

Proof. Joshua - TBD

4.7 Disjunctions and recursion in HBL

Using the syntax and semantics of HBL described so far, we can represent process-spawning chains with a fixed number of steps for the third motivating example from 2. The disjunctive and recursive extension defined in this section allows us to specify chains of unpredictable length using compact HBL watchpoints so that at any stage, *either* the current process spawns the target `powershell.exe`, *or else* it spawns a new process that becomes the current process for a recursive call.

The HBL language we have defined in Section 4.1 can be extended without unreasonable damage to Thm. 3. In particular, we consider two extensions, one containing disjunctions of HBL expressions, and the other containing disjunctions and also recursions. In the latter case, we will require that there is at most one recursive call in any one HBL expression.

hbl with disjunction. The syntax with disjunction is:

$$\begin{aligned} e ::= & \; a \\ & \mid \; a \rightarrow (e_1 \mid e_2 \mid \ldots \mid e_n) \\ & \mid \; a \rightsquigarrow \underline{b} \rightsquigarrow (e_1 \mid e_2 \mid \ldots \mid e_n) \end{aligned} \tag{8}$$

Thus, our previous syntax is simply the special case in which $n = 1$ always holds.

It is easy to update our execution model. In `process_one`, instead of generating an `apm` for a single resumption expression e, we generate a list of `apm`s for $e_1, e_2, \ldots, e_n$, which we label with the tag `Next`. In `process_records`, when `Next` delivers this list of `apm`s, we prepend its members before `so_far`.

We say that the *branch number* $\mathsf{bn}(a)$ of a SWE a is one, and the branch number $\mathsf{bn}(e)$ of a compound $e = a \rightarrow (e_1 \mid e_2 \mid \ldots \mid e_n)$ or $e = a \rightsquigarrow \underline{b} \rightsquigarrow (e_1 \mid e_2 \mid \ldots \mid e_n)$ is $\sum_{1 \leq i \leq n} \mathsf{bn}(e_i)$. Now it is clear that a single watchpoint can cause the generation of a number of `apm`s as it executes, but no more than its branch number. Therefore, in Thm. 3, we relax the bound on `apm`s by a factor of $\mathsf{bn}(e)$. This leads to:

Theorem 5 *Suppose that the algorithm of Fig. 4 is executed starting with a (fixed) set S of k watchpoints. Suppose that, for each $e \in S$, $\mathsf{bn}(e) \leq b$, and within those watchpoints at most time ℓ is needed to evaluate any single* SWE *on any record r at any stream position i.*

1. *Any call to* `process_one` *with a record r and active partial match a takes at most $2 \cdot \ell + bc$ time, where c characterizes the auxiliary functions used;*
2. *After processing i records in the stream, there are at most $i \cdot k \cdot b$ active partial matches;*
3. *For fixed k, the time needed for* `process_records` *to handle the i^{th} record is at most $\mathcal{O}(ib \cdot (\ell + b))$.*

Proof. Joshua -TBD

hbl with a recursive call. Suppose now that we extend the signature above with recursion variables X ranging over HBL expressions, and use the Xs to invoke recursive calls. The recursion variables are disjoint from the environment variables, and will be used in a different way.

Put definition around this stuff.

$$\begin{aligned} e ::= & \; a \quad \mid \quad X \\ & \mid \; a \rightarrow (e_1 \mid e_2 \mid \ldots \mid e_n) \\ & \mid \; a \rightsquigarrow \underline{b} \rightsquigarrow (e_1 \mid e_2 \mid \ldots \mid e_n) \end{aligned} \tag{9}$$

$$q ::= X = e \tag{10}$$

An equation q of the form $X = e$ says how to expand X when it is encountered, namely to continue by matching against e.

A watchpoint consists now of a sequence of equations $\langle X_i = e_i \rangle_{0 \le i \le k}$ where $0 \le k$. By convention, X_0 is the "main expression," and execution will start looking for a match to e_0. Any time an `apm` reaches a variable X_i, the processor looks it up in the equations and replaces it by its right hand side e_i.

We accept only watchpoints $W = (\langle X_i = e_i \rangle_{0 \le i \le k})$ satisfying two properties:

Guarded: Every a recursion variable occurrence X_i is in an $f_1, \ldots, f_n$ in an *after* expression $a \to (f_1 \mid f_2 \mid \ldots \mid f_n)$ or an *avoid* expression $a \rightsquigarrow \underline{b} \rightsquigarrow (f_1 \mid f_2 \mid \ldots \mid f_n)$; no occurrence replaces the lead a or avoid SWE b, and no e_j is identical with a recursion variable X_ℓ.

Linear: An expression e is *linear* iff it contains at most one variable occurrence. A watchpoint is *linear* iff every expression $e_0, \ldots, e_k$ in its equations is linear.

For guarded, linear watchpoints, there is not too large a runtime penalty. Let us define $\mathsf{bn}(W) = \max_{0 \le i \le k} \mathsf{bn}(e_i)$. Then a watchpoint with branch number b after one step may yield up to b `apm`s: invoking all its b continuations, one of which is a variable X_i that is replaced with an e_i. After another step this e_i may add another b `apm`s, so we now have $(b-1)+b$. One of the b fresh `apm`s is then expanded, and may after another step contribute a new b `apm`, yielding $2(b-1)+b$, etc. Since we add new copies of the k initial watchpoint at every step, this yields a bound in Clause (2) on the order of i^2bk, and in Clause (3) of $\mathcal{O}(i^2b \cdot (\ell + b))$. With a sliding window restriction, we can again replace i by the window width w.

If we relax the constraint that the watchpoint W be linear, we instead get a blow-up kb^i or kb^w that is exponential in i or w. This is why we require our watchpoints to be linear.

In the `powershell` example of Section 2, we have a "main expression" $X_0 = e_0$, where e_0 has the form $a \to c \mid X_1$. The SWE a looks for the creation of a process `pid` invoking the executable `Outlook.exe`. It stores `current` = `pid` into the environment that will be in place for the remainder. The first disjunct c of the remainder is an SWE that is satisfied by a process-spawn record, in which the parent is `current` and the child invokes the executable `powershell.exe`.

The other is a call to X_1. The body of X_1 is an after-expression $d \to c \mid X_1$. The SWE d is satisfied by any process-spawn record in which the parent is `current`. If the child process id is `pid`, it stores `current` = `pid` into the environment for its remainder. Its remainder has the same two cases c, X_1 as we saw in X_0. Each time through the recursion, the branch c succeeds if the current process invokes `powershell.exe`. The branch X_1 follows the chain of invocations one additional step, after which it is ready to succeed via c or continue following the chain.

4.8 Concurrency Analysis

Theorem 3 states that for a fixed number of watchpoints, k, the worst case time needed for `process_records` to handle the i^{th} record is $\mathcal{O}(i \cdot \ell)$. Let us explore whether the worst case time bound for evaluating HBL expressions can be reduced further with concurrent processing.

Lemma 6 *Suppose that the algorithm of Fig. 4 is executed starting with a (fixed) set S of k watchpoints, and within those watchpoints at most time ℓ is needed to evaluate any single* SWE *on any record r at any stream position i. With unlimited concurrency, the time to evaluate any single* SWE *on any record r at any stream position i at most $\mathcal{O}(k \cdot \ell)$ is needed.*

Proof. After processing i records in the stream, there are at most $i \cdot k$ active partial matches (from 2). Since a partial match does not depend on another partial match to continue its computation, every partial match can be evaluated concurrently. Therefore, for k watchpoints, $\mathcal{O}(k \cdot \ell)$ is the upper bound for evaluating any single SWE on any record r at any stream position i at most $\mathcal{O}(k \cdot \ell)$ is needed. □

Indeed, our Java implementation takes advantage of this possibility of concurrency. Fig. 4 differs from our Java implementation in two important ways. First, `List.fold_left` in OCaml handles each list entry sequentially, whereas our Java implementation can process a number of `apm`s concurrently when multiple threads are available. The standard version of OCaml does not offer multiple threads. Second, `prepend_apms` adds new (initial) active partial matches to the front of its second (list) argument; in fact we handle the `apm`s as a set in our Java implementation, so that we add members only if they are different from current members.

Realistically, however, unlimited concurrency is not a possibility. In many cases, unlimited concurrency is not required, and instead a thread can be spawned to process each partial match, and the number of threads spawned can be specified in HBL configuration. Let us say, the number of such threads is m. The worst case memory required to store the partial matches would be at worst $\mathcal{O}(m.k)$ at any time, and the computational upper bound reduces to $\mathcal{O}(k \cdot \ell/m)$.

On the other hand, if sliding windows are used, then the maximum concurrency required to achieve $\mathcal{O}(k \cdot \ell)$ limits to the number of records within the sliding window, and analytics writer can control the sliding window size for a given system, and thus improve performance.

5 Single-Width Expressions

The conclusions of the previous section are *parametric* in the choice of the SWE language and its semantics $\text{SAT}_\eta(a, \tau, i)$. In this section, we show that that evaluating the *satisfies* relation for a SWE, a, requires at most work proportional to the length of a (Lemma 1). That says that the work for any one active partial match to handle any new record has a predictable bound independent of the length of the input stream (or its contents).

An SWE consists of a *record specification* followed by a *binding clause*. The record specification determines whether any given record will successfully satisfy it. The binding clause takes effect only if the record specification succeeds, and it determines the resulting environment η' from η by saying which variable identifiers may differ from η, and what values η' yields for those variable identifiers. Fig. 5 states that a record may assert:

$$
\begin{aligned}
\textsc{swe} &::= \textsc{recsp}\ \textsc{bndsp} \\
\textsc{recsp} &::= \mathsf{absent}(\mathsf{fld}) \quad | \quad \mathsf{present}(\mathsf{fld}) \\
&\quad | \quad \mathsf{pred}(t^*) \quad | \quad \mathsf{not}(\textsc{recsp}) \\
&\quad | \quad \textsc{recsp}\ \mathsf{and}\ \textsc{recsp} \quad | \quad \textsc{recsp}\ \mathsf{or}\ \textsc{recsp} \\
\textsc{bndsp} &::= \langle \mathsf{empty} \rangle \quad | \quad \mathsf{letting}\ (\textsc{id} = t)^*
\end{aligned}
$$

Fig. 5: Abstract syntax for SWEs.

- a field fld is present or absent.
- a predicate pred holds of the values of terms t; a predicate may return True, False, or Never.
  A predicate returns Never when it can never again return true, given the point reached in the input stream, e.g. when the record index or timestamp is already too large. Individual predicate definitions determine which results are Never.
- a logical operation on values. The negation of Never is *true*; a conjunction of Never with anything returns Never; a disjunction all of whose components are Never yields Never; and a disjunction with non-Never components yields the same result as if the Never components were omitted.

A non-empty binding clause BNDSP consists of a sequence of *let*-bindings that associate identifiers ID with the values of terms t. New bindings may shadow earlier bindings to ID.

A term t may access the field values fld of the record $r = \tau \mathbin{@} i$, so as to remember them for use in evaluating future records $r' = \tau \mathbin{@} i'$, for instance to check whether they involve the same values. These field values may include timestamps from the generating systems. The term t may also involve the stream index i, so that evaluation of a future SWEs at a stream index j may depend on whether $j - i$ has surpassed a bounded *window-length*. This is a prime reason that SAT returns Never, since no subsequent $j - i$ will decrease. Timestamps may also be used to generate Never responses, since it may be reasonable to conclude that the difference between timestamps will never decrease beyond a limited jitter amount.

The implemented HBL provides a rich language for building terms t and predicates pred. Its concrete syntax is designed with user expectations in mind, unlike the abstract syntax of Fig. 5. In effect, it builds in some bookkeeping operations into the binding clauses, so that—for instance—a final successful environment always contains a list of the stream indices at which successive SWEs a were satisfied.

SWE processing bound. Based on a number of reasonable assumptions, we can conclude that the compute time needed to evaluate a SWE is linear in its length as an expression, i.e. the total number of symbols in it. In particular, we assume that there is a fixed maximum cost to evaluate any of the following:

a single field from a record, or to observe the absence of a field, regardless of record type.

a single value from the environment η, i.e. to retrieve $\eta(x)$ for any identifier x.

an update for an environment entry $\eta(x)$ for an identifier x, once the new value v has been evaluated.

a single function symbol $f(v_1, \ldots, v_k)$ that may appear in a binding clause, once its k arguments have been evaluated.

the truth value of a predicate $\mathsf{pred}(v_1, \ldots, v_k)$ once its k arguments have been evaluated.

Assuming that no individual record type has fields of genuinely unpredictable length, familiar data structures (e.g. vectors for environments) allow an implementation to satisfy these conditions.

Lemma 1. *On the five assumptions just mentioned, given any record $r = \tau \mathbin{@} i$ and any* SWE *a where the length of a is ℓ, computing $\mathrm{SAT}_\eta(a, \tau, i)$ can be done in time $\mathcal{O}(\ell)$.*

Proof. By induction on the structure of a. Each clause in the grammar of Fig. 5 increases the length of its arguments; the assumptions ensure that evaluating SAT adds only a constant amount to the cost of evaluating the arguments.

Combining Lemma 1 with Thm. 3 we may infer:

Corollary 7 *Suppose that the algorithm of Fig. 4 is executed starting with a (fixed) set S of k watchpoints, none of which contains an* SWE *of length greater than the fixed value ℓ_0.*

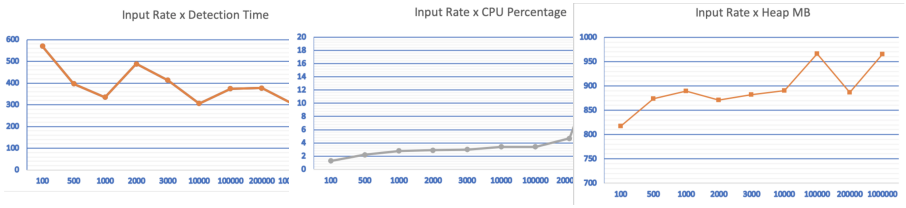
Then the time complexity needed for `process_records` *to handle the i^{th} record is at most $\mathcal{O}(i)$.*

Moreover, as we noted after Thm. 3, this bound is often unnecessarily pessimistic, since the population of active partial matches tends to grow much more slowly than the worst case bound of $i \cdot k$ in Thm. 3, clause 2. Indeed, our performance measurements in Section 6 agree. We formalize the window-bounded case:

Corollary 8 *In addition to the assumptions of Cor. 7, assume each watchpoint $e_j \in S$ enforces a window w_j, with each $w_j < w$. The time complexity for* `process_records` *to handle any record is $\mathcal{O}(w)$.*

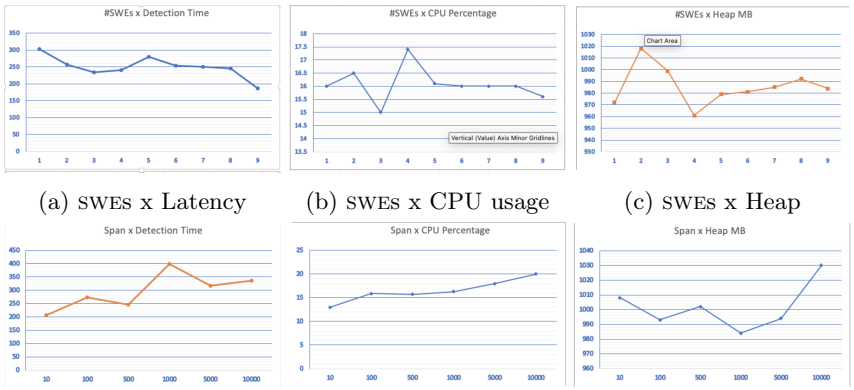
6 Sensitivity Analysis

Our goal in this section is to report the results of experiments conducted to evaluate the sensitivity of our algorithm to various operational parameters based on its implementation in Java. Conducting absolute performance measurements is not our goal, since the performance could vary based on the software and hardware used to implement the algorithm. Karimov et al. [18] define a processing-time latency metric as the interval between a tuple's ingestion time (i.e., the time that



(a) Rate x Latency (b) Rate x CPU usage (c) Rate x Heap

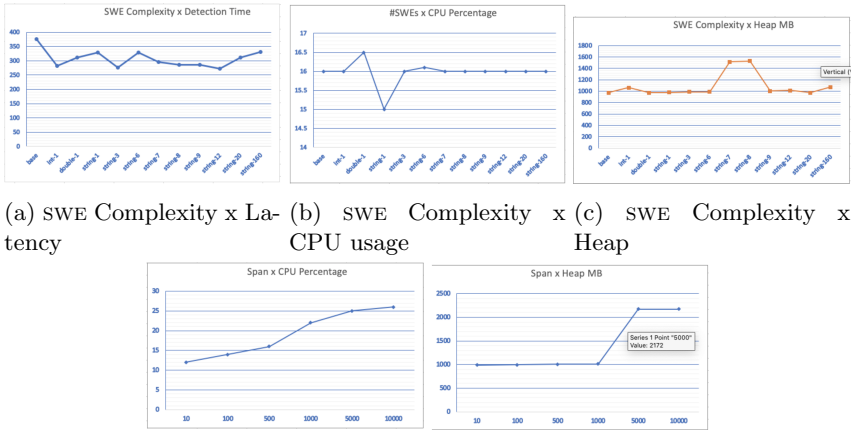
Fig. 6: Sensitivity Analysis for Input Arrival Rate



(a) SWEs x Latency (b) SWEs x CPU usage (c) SWEs x Heap

(d) Sliding Window x Latency (e) Sliding Window x CPU usage (f) Sliding Window x Heap

Fig. 7: SWEs and Sliding Window



(a) SWE Complexity x Latency (b) SWE Complexity x CPU usage (c) SWE Complexity x Heap

(d) Evidence x CPU usage (e) Evidence x Heap

Fig. 8: SWE Complexity and Evidence

the event has reached the input operator of the streaming system) and the emission of an alert from the streaming analytics engine. For aggregate operations such as *avg*, [18] defines process-time latency as the *maximum* processing-time of all events that contributed to that event (or alert in HBL). The calculation of *maximum* processing-time is irrelevant in an HBW with multiple SWEs, since the most relevant time is the ingestion time of the last event record that will complete the *avg*, and and generate the average value as an alert. Therefore, we measure the time difference between the arrival into the HBL Processor of the tuple or event record that matches the last SWE in a watchpoint, and the generation of the alert. We also measured the changes in computing resource usages during these experiments.

In order to compare latency by varying different parameters in controlled conditions, we chose to artificially generate event records. The operational parameters that we individually varied were the following: the rate of arrival of events, predicate depth or the number of SWEs in a watchpoint, the size of sliding window, size of event type name, the average size of event record, and the operator complexity of SWE. The default values used in the experiments were {arrival rate: 1 million per second, matching span:10000 records, predicate length: 2, SWE complexity: 2}.

When the rate of arrival of events were varied from 100 to 1 million per second, the process-time latency remained within 600 to 250 microseconds (Fig. 6a), whereas the CPU usage percentage changed from under 6% to slightly over 18% Fig. 6b, and the heap memory used varied from around 1GB to 2GB (Fig. 6c). The increase in CPU usage and heap memory is expected because more number of records are processed with increased rate of arrival.

The number of SWEs in a watchpoint was increased from 1 to 9, and the latency was measured. The latency remained roughly within 300 microseconds to 200 microseconds (Fig. 7a). The CPU usage was between 15% to 17.5% (Fig. 7b), with the heap memory in the range of 960MB to 1020MB (Fig. 7c).

We varied the size of the sliding windows. We predicted that larger sliding windows would increase the usage of the computational resources (Theorem 3). When we increased the size from 10 to 10000, the latency remained within 200 to 400 microseconds (Fig. 7d), whereas the CPU usage steadily increased from 13% to 20% (Fig. 7e), and the heap usage increased from 980MB and 1030MB (Fig. 7f). These analytics did not return the event stream segments that matched, only an alert when a match occurred. When the experiments were repeated with alerts that retuned event stream segments, referred to as Evidence, while the CPU usage (Fig. 8d) was comparable to (Fig. 7e), heap usage increased considerably (Fig. 8e) compared to Fig. 7f, as expected, since the partially matched records were stored in memory.

The impact of the complexity of operations and operators were evaluated with string, double and int operations with variable number of operators ranging from 1 to 160. While Lemma 1 predicts increased cost with increased length of expressions, our Java implementation did not show much sensitivity to this length (Fig. 8a), or CPU usage (Fig. 8b) which leads us to think that the com-

putational cost of SWE evaluation is less significant than the cost of maintaining partial matches.

The detection algorithm was implemented in Java, and run with Java HotSpot™ 64-bit server VM with no optimization settings, on a Macbook Pro, 2.2GHz, 16GB. The experiments were run from an Eclipse IDE, and the computing resource usage was measured with VisualVM [27]. The Java based implementation had a size of 11MB, including all the jars. The implementation included a compiler for HBL written using ANTLR4 [24] and the HBL Processor. The HBL compiler generated object code in JSON format that was used for generating alerts.

7 Related Work

Cugola and Margara [7] trace the evolution of modern streaming analytics systems from active database systems and classic Complex Event Processing (CEP) systems that permit complex queries. A distinguishing feature of streaming analytics systems from classical CEP based querying systems is the need to perform real-time or quasi-real-time processing of incoming information to produce new knowledge [7]. CEP is also applicable to Big Data processing as detailed in this survey [11]. Event Stream Processing (ESP) is a related area of research that entirely focuses on processing event streams, as exemplified by the design patterns described in this tutorial [25]. A recent survey by Dayarathna and Perera [9] attempts to generalize ESP systems and CEP systems under a common Event Processing (EP) umbrella.

The languages used by streaming analytics systems fall into two categories: SQL like in varying degrees, or non-SQL. One of the early SQL-like category of query languages is StreamSQL [31], which eventually led to the development of a streaming analytics system in a commercial platform that uses a visual programming language called EventFlow, which allows developers to rapidly create and tailor stream processing applications and connect to over 150 streaming and static data sources with its pre-built data connectivity integration points. IBM has a competing StreamBase system that compares well with TIBCO's system [14]. Both the TIBCO and IBM systems are primarily intended for business analytics for event streams such as stock ticker symbols. One of the early non-SQL like languages for streaming analytics is Event Processing Language (EPL) used by Esper [22,30]. Since Esper is written in Java, it is capable of triggering alert actions directly in Java code.

The key language features of streaming analytics languages are: causality operator, time and tuple-based based sliding window specification, stateful computation, aggregation operators, and event attribute predicate expressions. The support for these features vary in different streaming analytics languages. Several alternate references to the Happened-Before operator exist such as the *SEQ* operator in SASE query language [34], and *followed-by* operator in Apache Flink [5]. This operator specifies that one event appears in an event stream prior to another event, with the implied assumption that the earlier occurring event

will arrive first in an event stream. The sliding window is used for continuous evaluation, and was applied to streams first by the Aurora system [6]. Stateful computing is used to correlate information across multiple events that occur spatially separated in a space or time window. Logical expressions with event attributes are used to identify the events with the right attributes. While the basic concepts of these features remain the same, there can be much variation in how these features are supported, and the specifics of the support.

Since real-time is a key goal of streaming analytics, reducing latency and increasing throughput are of much interest. A recent benchmark of three open source streaming analytic systems, Apache Flink [5], Apache Spark [35], and Apache Storm [32] finds that if the average latency is a priority, then Flink is the better choice [18]. A hardware based design using FPGA for a C-based event language is reported to be capable of processing with a 20Gbps network interface card [15].

Another aspect of streaming analytics systems is the use of a data streaming model for further processing the outputs of streaming analytics systems using additional streaming analytics engines. Akdere et al. [1] discuss a data flow streaming model. IBM's SPL is another approach to managing the data streams between the analytic engines [14]. Apache Flink [5] is an open source system that also utilizes a data flow streaming model. When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks. The dataflows resemble arbitrary directed acyclic graphs (DAGs)

When event records have timestamps, we assume their values in the input stream are in monotonically non-decreasing order, or nearly so. Records may also have fields similar to timestamps, such as Mattern's vector clock values [**?**]. An important question any EP system will face is the ordering of events in the event stream, especially since events do not occur instantaneously, and thus events overlap each other. Indeed, segments in the input event stream may be considered *composite events*, and composite events certainly have duration. White et al. [**?**] address the ordering of events, and define a successor function that can be implemented to order an event stream. Therefore, we assume when an input event has a time stamp that specifies its time of occurrence, it is always possible to find the successor of that event in a stream.

If records have position identifiers embedded in them, we assume they are strictly increasing in the event stream. Even if records do not have a timestamp or a positional index field embedded within them, they arrive in a sequence. Thus, our processing may depend on the numerical position of a record within the input event stream.

[5].

7.1 Comparison to SAQL

Streaming analytics have been applied for real-time detection of security incidents for a number of years [12,16,4]. Recently, Gao et al. developed SAQL, a domain specific language, for specifying abnormal system behaviors [13]. SAQL

is supported by an efficient query generator and scheduler over Siddhi streaming analytics engine [29,28]. SAQL queries can specify rule-based anomalies, time-series anomalies, invariant-based anomalies, and outlier-based anomalies. SAQL is targeted for deployment in an enterprise environment with less than 2s latency and scalable throughput. An alternate search-based approach for time-series queries is by using Splunk [3]. Splunk needs to index the incoming events using time stamps, store the the events and indices to support or searching. While this approach may be useful for analysis in a Security Operations Center (SOC) of an enterprise, it still lacks the ability to provide near real-time detection by systems such as SAQL and HBL.

Since SAQL is specifically targeted for use in detection of cyber effects, and it is a recent language, we are going to do a more detailed comparison with SAQL here. The first difference between SAQL and HBL is that HBL's streaming analytics engine is implemented using the algorithm described in the previous sections, whereas, SAQL has a query engine that is built over an existing Siddhi streaming analytics engine. Event patterns in SAQL supports an event type using a subject-object entity, whereas HBL supports event types and sets of event types, and does not have any constraints on what constitutes an event type, allowing detection of event occurrences that occur in any order in time. Event attribute relationship in SAQL is supported in HBL using variables in the execution context. SAQL supports an event temporal relationship similar to Happened-Before relationship. HBL supports comprehensive attribute expressions, similar to SAQL. SAQL supports sliding windows in time, whereas HBL supports sliding windows in time and space, i.e., number of records. HBL and SAQL support rule based anomalies as well as time-series anomalies using the execution context to store the time series operators such as average. SAQL provides specialized language constructs to store information from prior

8 Conclusions

In this paper, we introduce a new algorithm for Event Processing (EP) systems with an upper bound for latency. This algorithm was implemented using a new language, Happened-Before Language (HBL), and its processor. We provide the formal semantics for HBL, and we show that the algorithm has an upper bound with a linear complexity relative to the size of the sliding window size. The analytics in HBL, specified in an HBL watchpoint contains one or more Single Width Expressions (SWEs) separated by the Happened-Before operator. We evaluated the sensitivity of the algorithm to the event arrival rate in the event stream, number of SWEs in a watchpoint, distance between the first matching event to the first SWE in a watchpoint to the last event matching the watchpoint, and the complexity of SWE itself. The results show that as expected, process-time latency remains within a bound, while the CPU utilization and heap memory usage increase with increased complexity in some cases, expect in the case of number of SWEs in an HBL watchpoint, where there is no significant increase.

We developed our approach to event processing for edge devices and end-points where relatively reduced computational resources are available.

The edge systems are located in remote locations, and often their operators are not necessarily experts in cyber attack detection. Therefore, the EP system at the edge should only support analytics that must result in computations guaranteed to terminate in a reasonable time [26]. While a general purpose streaming analytics language cannot prevent the analytics designers from creating inefficient and nonterminating computations, it is possible for a language designer to expose to the analytics designer simple paradigms such as sliding windows that reduce the computational load, and the detection engine to implement time based limits to prevent runaway threads from consuming computational resources. The discussion of such ease of use and run-time optimization features in HBL are beyond the scope of this paper.

```
type record = string list

type e_val = St of string
  | N of int
  | L of e_val list

type env = (string * e_val) list

type io_struct = {
  get     : (unit -> record);
  report  : (env -> unit);
  index   : (unit -> int)
  }

type sat =
  | E of env
  | False
  | Never

type swe = record -> int

type exp = | S of swe
           | Arr of swe * exp
           | Av of swe * swe * exp

let lead = function
  S a                -> a
  | Arr (a,_)        -> a
  | Av (a,_,_)       -> a

let remainder = function
  | S a              -> None
  | Arr (_,e)        -> Some e
  | Av (_,_,e)       -> Some e

let avoid = function
  | S _              -> None
  | Arr (_,_)        -> None
  | Av (_,b,_)       -> Some b

type apm = { sat: exp;
             avoider : swe option ;
             env : env }

let eta_0 = []    (* initial env empty *)

let init_apm e i = {
  e = e ;
  avoider = None ;
  env = eta_0 }

let make_apm e prev_e eta = {
  e = e ;
  avoider = avoid prev_e;
  env = eta }

let rec prepend_apms i l = function
  | []          -> l
  | e :: rest   ->
    prepend_apms i ((init_apm e i) :: l) rest

type spec_res =
  | Success of env
  | Next of apm
  | Fail
```

Fig. 9: Auxiliary OCaml definitions

References

1. Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008.

2. Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
3. Michael Joseph Baum, R David Carasso, Robin Kumar Das, Rory Greene, Bradley Hall, Nicholas Christian Mealy, Brian Philip Murphy, Stephen Phillip Sorkin, Andre David Stechert, Erik M Swan, et al. Search based on a relationship between log data and data from a real-time monitoring environment, August 29 2017. US Patent 9,747,316.
4. Lars Baumgärtner, Christian Strack, Bastian Hoßbach, Marc Seidemann, Bernhard Seeger, and Bernd Freisleben. Complex event processing for reactive security monitoring in virtualized computer systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 22–33. ACM, 2015.
5. Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
6. Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.
7. Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.
8. Suresh K Damodaran and Joshua D Guttman. Systems and methods for declarative specification, detection, and evaluation of happened-before relationships, December 31 2019. US Patent 10,521,331.
9. Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, 51(2):33, 2018.
10. Richard Dedekind. Was sind und was sollen die zahlen? Brunschweig, 1888. Translated as "The Nature and Meaning of Numbers," in Richard Dedekind, *Essays on the Theory of Numbers*, New York: Dover, 1963.
11. Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217–236, 2017.
12. Ruediger Gad, Martin Kappes, Juan Boubeta-Puig, and Inmaculada Medina-Bulo. Employing the cep paradigm for network analysis and surveillance. In *Proceedings of the Ninth Advanced International Conference on Telecommunications*, pages 204–210. Citeseer, 2013.
13. Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. {SAQL}: A stream-based query system for real-time abnormal system behavior detection. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 639–656, 2018.
14. Martin Hirzel, Scott Schneider, and Buğra Gedik. Spl: An extensible language for distributed stream processing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(1):5, 2017.
15. Hiroaki Inoue, Takashi Takenaka, and Masato Motomura. Hardware design for c-based complex event processing. In *Embedded Systems Design with FPGAs*, pages 79–100. Springer, 2013.

16. Keerthi Jayan and Archana K Rajan. Preprocessor for complex event processing system in network security. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 187–189. IEEE, 2014.
17. Michael Jones. Spoofing in the black sea: What really happened? `https://www.gpsworld.com/spoofing-in-the-black-sea-what-really-happened/`. Accessed: 2020-02-15.
18. Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.
19. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
20. Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml programming language. `https://ocaml.org`. Accessed 18 Feb. 2020.
21. Jared Myers and Cathy Cramer. Carbon black tau threat analysis: Emotet banking trojan leverages ms office word docs, powershell to deliver malware. `https://www.carbonblack.com/2018/06/04/carbon-black-tau-threat-analysis-emotet-banking-trojan-leverages-ms-office-word-docs-powershell-d` Accessed: 2019-07-15.
22. Jared Myers and Cathy Cramer. Complex event parsing with esper – core concepts. `https://medium.com/@bruno.felix/complex-event-processing-with-esper-core-concepts-f97394b39c07`, 2017. Accessed: 2019-07-15.
23. Youngseok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.
24. Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
25. Adrian Paschke, Paul Vincent, Alex Alves, and Catherine Moxey. Tutorial on advanced design patterns in event processing. In *Proceedings of the 6th acm international conference on distributed event-based systems*, pages 324–334. ACM, 2012.
26. Simone Santini. Querying streams using regular expressions: some semantics, decidability, and efficiency issues. *The VLDB Journal—The International Journal on Very Large Data Bases*, 24(6):801–821, 2015.
27. Jiri Sedlacek and Tomas Hurka. Visualvm. `https://visualvm.github.io/`, 2008-2020. Accessed: 2020-02-20.
28. Team Siddhi. Siddhi: Stream processing and complex event processing engine. `https://github.com/siddhi-io/siddhi`. Accessed: 2019-07-15.
29. Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pages 43–50. ACM, 2011.
30. Esper Team. Espertech inc. esper reference version 8.2.0, 2019.
31. StreamSQL Team. Streamsql: a data stream language extending sql. `https://en.wikipedia.org/wiki/StreamSQL`. Accessed: 2019-07-15.
32. Jan Sipke van der Veen, Bram van der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J Meijer. Dynamically scaling apache storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 154–161. IEEE, 2015.

33. Louis Woods, Jens Teubner, and Gustavo Alonso. Complex event detection at wire speed with fpgas. *Proceedings of the VLDB Endowment*, 3(1-2):660–669, 2010.
34. Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.
35. Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.