Streaming Process Mining

Andrea Burattin(✉)

Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
andbur@dtu.dk

Abstract. Streaming process mining refers to the set of techniques and tools which have the goal of processing a stream of data (as opposed to a finite event log). The goal of these techniques, similarly to their corresponding counterparts described in the previous chapters, is to extract relevant information concerning the running processes. This chapter presents an overview of the problems related to the processing of streams, as well as a categorization of the existing solutions. Details about control-flow discovery and conformance checking techniques are also presented together with a brief overview of the state of the art.

Keywords: Streaming process mining · Event stream

1 Introduction

Process mining techniques are typically classified according to the task they are meant to accomplish (e.g., control-flow discovery, conformance checking). This classification, though very meaningful, might come short when it is necessary to decide which algorithm, technique, or tool to apply to solve a problem in a given domain, where nonfunctional requirements impose specific constraints (e.g., when the results should be provided or when the events are recorded).

Most algorithms, so far, have been focusing on a static event log file, i.e., a finite set of observations referring to data collected during a certain time frame in the past (cf. Definition 1 [1]). In many settings, however, it is necessary to process and analyze the events as they happen, thus reducing (or, potentially, removing) the delay between the time when the event has happened in the real world and when useful information is distilled out of it. In addition, the amount of events being produced is becoming so vast and complex [22] that storing them for further processing is becoming less and less appealing. To cope with these issues, event-based systems [4] and event processing systems [19] can become extremely valuable tools: instead of storing all the events for later processing, these events are immediately processed and corresponding reactions can be taken immediately. In addition, event-based systems are also responsive systems: this means they are capable of reacting autonomously when deemed necessary (i.e., only when new events are observed).

Coping with the above-mentioned requirements in the context of data analysis led to the development of techniques to analyze streams of data [3,21].

A data stream is, essentially, an unbounded sequence of observations (e.g., events), whose data points are created as soon as the event happens (i.e., in real-time). Many techniques have been developed, over the years, to tackle different problems, including frequency counting, classification, clustering, approximation, time series analysis and change diagnosis (also known as novelty detection or concept drift detection) [20,46]. Process mining techniques applied to the analysis of data streams fall into the name of *streaming process mining* [8] and both control-flow discovery as well as conformance checking techniques will be discussed later in the chapter.

The rest of this chapter is structured as follows: this section presents typical use cases for streaming process mining and the background terminology used throughout the chapter. Section 2 presents a possible taxonomy of the different approaches for streaming process mining, which can be used also to drive the construction and the definition of new ones. Section 3 introduces the problem of streaming process discovery, by presenting a general overview of the state of the art and the details of one algorithm. Section 4 sketches the basic principles of streaming conformance checking. As for the previous case, also this section starts with a state-of-the-art summary and then dives into the details of one algorithm. Section 5 mentions other research endeavors of streaming process mining and then concludes the chapter.

1.1 Use Cases

This subsection aims at giving an intuition of potential use cases for streaming process mining. In general, every setting that requires drawing conclusions before the completion of a running process instance is a good candidate for the application of streaming process mining. In other words, streaming process mining is useful whenever it is important to understand running processes rather than improving *future* ones or "forensically" investigate those *from the past*.

Process discovery on event streams is useful in domains that require a clear and timely understanding of the behavior and usage of a system. For example, let's consider a web application to self-report the annual tax statement for the citizens of a country. Such a system, typically, requires a lot of data to be inserted over many forms and, usually, the majority of its users have to interact with help pages, FAQs, and support information. In this case, it might be useful to understand and reconstruct the flow of a user (i.e., one process instance) to understand if they are getting lost in a specific section, or in specific cycles and, if necessary, provide tailored help and guidance support. Since the ultimate goal is to improve the running process instances (i.e., helping the users currently online), it is important that the analyses process the events immediately and that corresponding reactions are implemented straight away.

Conformance checking on event streams is useful whenever it is important to immediately detect deviations from reference behavior to enact proper countermeasures. For example, let's consider operational healthcare processes [31], in most of these cases (in particular in the case of non-elective care processes, such as urgent or emergency ones) it is critically important to have a sharp understanding of each individual patient (i.e., one process instance), their treatment evolution, as well as how the clinic is functioning. For example, when treating a patient having acute myeloid leukemia it is vital to know that the treatment is running according to the protocol and, if deviations were to occur, it is necessary to initiate compensation strategies immediately.

Another relevant example of streaming conformance checking could derive from the investigation of the system calls of the kernel of an operating system when used by some services or applications. These calls should be combined in some specific ways (e.g., a file should be `open()`, then either `write()` or `read()` or both appear, and eventually the file should be `close()`) which represent the reference behavior. If an application is observed strongly violating such behavior it might be an indication of strange activities going on, for example trying to bypass some limitations or privileges. Exactly the same line of reasoning could be applied when consuming RESTful services.

Additional use cases and real scenarios are depicted in the research note [5], where real-time computing[1], to which streaming process mining belongs, is identified as one of the impactful information technology enablers in the BPM field.

1.2 Background and Terminology

This section provides the basic background on streams as well as the terminology needed in the rest of the chapter.

A stream is a sequence of observable units which evolves over time by including new observations, thus becoming unbounded[2]. An *event stream* is a stream where each observable unit contains information related to the execution of an event and the corresponding process instance. In the context of this chapter, we assume that each event is inserted into the stream when the event itself happens in the real world. The universe of observable units $\mathcal{O}$ can refer to the activities executed in a case, thus having $\mathcal{O} \subseteq \mathcal{U}_{act} \times \mathcal{U}_{case}$ (cf. Definition 1 [1]), as discussed in Sect. 3) or to other properties (in Sect. 4 the observable units refer to relations B between pairs of activities, i.e., $\mathcal{O} \subseteq (B \times \mathcal{U}_{act} \times \mathcal{U}_{act}) \times \mathcal{U}_{case}$).

Definition 1 (Event stream). *Given a universe of observable units $\mathcal{O}$, an event stream is an infinite sequence of observable units: $S : \mathbb{N}_{\geq 0} \to \mathcal{O}$.*

We define an operator *observe* that, given a stream S, it returns the latest observation available on the stream (i.e., $\mathit{observe}(S) \in \mathcal{O}$ is the latest observable unit put on S).

[1] Please note that the paper explicitly mentions that, in that context, "*real-time computing refers to the so-called near real-time, in which the goal is to minimize latency between the event and its processing so that the user gets up-to-date information and can access the information whenever required*", thus perfectly matching our notion of streaming process mining.

[2] Please note that, in the literature, it is possible to distinguish other streaming models, where elements are also deleted or updated [21]. However, in this chapter we will assume an "insert-only model".

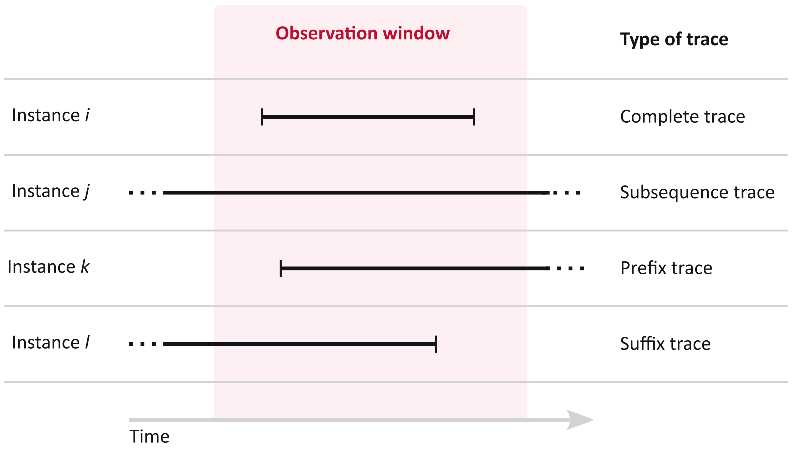
Due to the nature of streams, algorithms designed for their analyses are required to fulfill several constraints [6,7], independently from the actual goal of the analyses. These constraints are:

– it is necessary to process one event at a time and inspect it at most once;
– only a limited amount of time and memory is available for processing an event (ideally, constant time and space computational complexity for each event);
– results of the analyses should be given at any time;
– the algorithm has to adapt to changes over time.

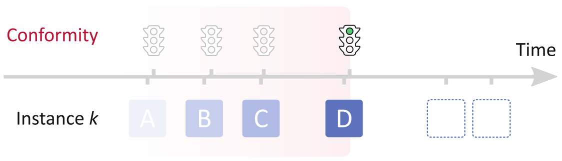
As detailed in [21, Table 2.1], it is possible to elaborate on the differences between systems consuming data streams and systems consuming static data (from now on, we will call these "offline"): in the streaming setting, the data elements arrive incrementally (i.e., one at the time) and this imposes the analysis to be incremental as well. These events are transient, meaning that they are available for a short amount of time (during which they need to be processed). In addition, elements can be analyzed at most once (i.e., no unbounded backtracking), which means that the information should be aggregated and summarized. Finally, to cope with concept drifts, old observations should be replaced by new ones: while in offline systems, all data in the log is equally important, when analyzing event stream the "importance" of events decreases over time.

In the literature, algorithms and techniques handling data streams are classified into different categories, including "online", "incremental", and "real-time". While *real-time systems* are required to perform the computation within a given deadline – and, based on the consequences of not meeting the deadline, they are divided into hard/soft/firm –, *incremental systems* just focus on processing the input one element at the time with the solution being updated consequently (no emphasis/limit on the time). An *online system* is similar to an incremental one, except for the fact that the extent of input is not known in advance [37]. Please note that, in principle, both real-time and online techniques can be used to handle data streams, thus we prefer the more general term *streaming techniques*. In the context of this chapter, the streaming techniques are in between the family of "online" and "soft real-time": though we want to process each event *fast*, the notion of deadline is not always available, and, when it is, missing it is not going to cause a system failure but just degradation of the usefulness of the information.

When instantiating the streaming requirements in the process mining context, some of the constraints bring important conceptual implications, which are going to change the typical way process mining is studied. For example, considering that each event is added to the stream when it happens in the real world means that the traces we look at will be incomplete most of the time. Consider the graphical representation reported in Fig. 1a, where the red area represents the portion of time during which the streaming process mining is active. Only in the first case (i.e., instance i), events referring to a complete trace are seen. In all other situations, just incomplete observations are available, either because events happened before we started observing the stream (i.e., instance l, suffix trace) because the events have still to happen (i.e., instance k, prefix trace),



(a) Different types of (incomplete) traces w.r.t. the observation window.



(b) Provide immediate feedback and keep it up-to-date with respect to the input.

Fig. 1. Process mining implications of some streaming requirements.

or because of both (i.e., instance j, subsequence trace). Figure 1b is a graphical representation of what it means to give results at any time in the case of conformance: after each event, the system needs to be able to communicate the conformity value as new events come in. Also, the result might change over time, thus adapting the computation to the new observations.

2 Taxonomy of Approaches

Different artists are often quoted as saying: “Good artists copy, great artists steal”. Regardless of who actually said this first[3], the key idea is the importance of understanding the state of the art to incorporate the key elements into newly designed techniques. Streaming process mining techniques have been around for some years now, so it becomes relevant to understand and categorize them in order to relate them to each other and derive new ones.

[3] Many people, including Pablo Picasso, William Faulkner, Igor Stravinsky, and several others are often referred to as the “first author” of some version of the quote. Actually, investigating the history of this quote on the Internet represents a formative yet very procrastination-prone activity (see also https://xkcd.com/214/).

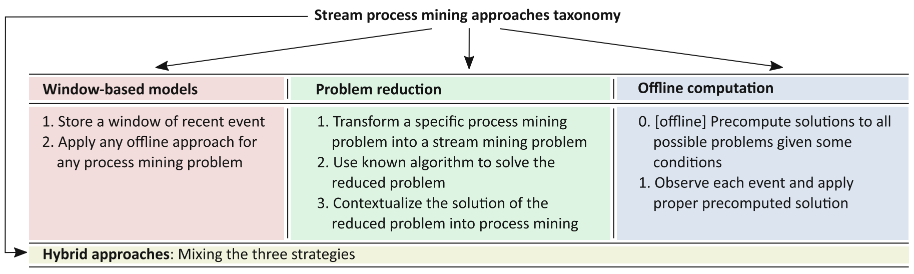

Stream process mining approaches taxonomy
Window-based models
1. Store a window of recent event
2. Apply any offline approach for any process mining problem
Problem reduction
1. Transform a specific process mining problem into a stream mining problem
2. Use known algorithm to solve the reduced problem
3. Contextualize the solution of the reduced problem into process mining
Offline computation
0. [offline] Precompute solutions to all possible problems given some conditions
1. Observe each event and apply proper precomputed solution
Hybrid approaches: Mixing the three strategies

Fig. 2. Taxonomy of the different approaches to solve the different stream process mining problems. For each technique, corresponding general steps are sketched [10].

It is possible to divide the currently available techniques for streaming process mining into four categories. A graphical representation of such taxonomy is available in Fig. 2, where three main categories are identified, plus a fourth one, which represents possible mixes of the others. In the remainder of this section, each category will be briefly presented.

Window Models. The simplest approach to cope with infinite streams consists of storing only a set of the most recent events and, periodically, analyzing them. These approaches store a "window" of information that can be converted into a static log. Then, standard (i.e., offline) analyses can be applied to the log generated from such a window. Different types of windowing models can be used and classified based on how the events are removed [34]. These models can be characterized along several dimensions, including the unit of measurement (i.e., whether events are kept according to some logical or physical units such as the time of the events or the number of events); the edge shift (so whether any of two bounds of a window is fixed to a specific time instant or if these change over time); and the progression step (i.e., how the window evolves over time, assuming that either of the bounds advances one observation at a time or several data points are incorporated at once). These profiles can create different window models, such as:

– count-based window: at every time instance, the window contains the latest N observations;
– landmark window: one of the window bounds is fixed (e.g., the starting time or the end time, far in the future) whereas the other progresses with time;
– time-based sliding window: in this case, the sliding window progresses according to two parameters: a time window duration and a time progression step;
– time-based tumbling window: similar to the time-based sliding window but where entire batches of events are observed so that two consecutive windows do not overlap.

Algorithm 1: Count-based window model process mining algorithm

```
Input: $S$: event stream
       $M$: memory
       $max_M$: number of observation to keep
       $A$: additional information (e.g., a reference model), can be $\emptyset$
1  forever do
       // Observe a new event
2      $e \leftarrow observe(S)$
       // Memory update
3      if $max(M) \geq max_M$ then
4          $dequeue(M)$ // Forgetting
5      end
6      $insert(M, e)$
       // Mining update
7      if perform mining then
           // Memory into event log
8          $L \leftarrow convert(M)$
9          $ProcessMining(L, A)$
10     end
11 end
```

Algorithm 1 reports a possible representation of an algorithm for process mining on a count-based window model. The algorithm uses as a memory model a FIFO queue and it starts with a never-ending loop which comprises, as the first step, the observation of a new event. After that, the memory is checked for maximum capacity and, if reached, the oldest event observed is removed. Then, the mining can take place, initially converting the memory into a process mining capable log and then running the actual mining algorithm on the given log.

Window-based models come with many advantages such as the capability of reusing any offline process mining algorithm already available for static log files. The drawback, however, comes from the inefficient handling of the memory: window-based models are not very efficient for summarizing the stream, i.e., the logs generated from a window suffer from strong biases due to the rigidity of the model.

Problem Reduction. To mitigate the issues associated with window models, one option consists of employing a problem reduction technique (cf. Fig. 2). In this case, the idea is to *reduce* the process mining problem at hand to a simpler yet well-studied problem in the general stream processing field in order to leverage existing solutions and derive new ones. An example of a very well studied problem is *frequency counting*: counting the frequencies of variables over a stream (another example of a relevant and well-studied problem is sampling). To properly reduce a process mining problem to a frequency counting one, it is important

Algorithm 2: Lossy Counting

```
Input: $S$: data stream
       $\epsilon$: maximal approximation error
1  $T \leftarrow \emptyset$ // Initially empty set
2  $N \leftarrow 1$ // Number of observed events
3  $w \leftarrow \lceil \frac{1}{\epsilon} \rceil$ // Bucket width
4  forever do
5      $e \leftarrow observe(S)$
6      $b_{curr} \leftarrow \lceil \frac{N}{w} \rceil$
       // Is there a tuple in $T$ with $e$ as first component?
7      if $e$ is already in $T$ then
8          Increment the frequency of $e$ in $T$
9      else
10         Insert $(e, 1, b_{curr} - 1)$ in $T$
11     end
12     if $N \mod w = 0$ then
13         forall the $(a, f, \Delta) \in T$ s.t. $f + \Delta \leq b_{curr}$ do
14             Remove $(a, f, \Delta)$ from $T$
15         end
16     end
17     $N \leftarrow N + 1$
18 end
```

to understand what a variable is in the process mining context and if it is indeed possible to extract information by counting how often a variable occurs.

An algorithm to tackle the frequency counting problem is called Lossy Counting [30], described in Algorithm 2 and graphically depicted in Fig. 3. Conceptually, the algorithm divides the stream into "buckets", each of them with a fixed size (line 6). The size of the bucket is derived from one of the inputs of the algorithm ($\epsilon \in [0, 1]$) which indicates the maximal acceptable approximation error in the counting. Lossy Counting keeps track of the counting by means of a data structure T, where each component (e, f, Δ) refers to the element e of the stream (the variable to count), its estimated frequency f, and the maximum number of times it could have occurred Δ (i.e., the maximum error). Whenever a new event is observed (line 5), if the value is already in the memory T, then the counter f is incremented by one (line 8), instead, if there is no such value, a new entry is created in T with the value e corresponding to the observed variable, frequency $f = 1$, and maximum error equal to the number of the current bucket minus one (from here it is possible to understand that since the buck size depends on the maximum allowed error, the higher the error, the larger the bucket size and hence the higher the approximation error) (line 10). With a fixed periodicity (i.e., every time a new conceptual bucket starts) the algorithm cleans the memory, by removing elements not frequently observed (lines 12–16). Please note that this specific algorithm has no memory bound: the size

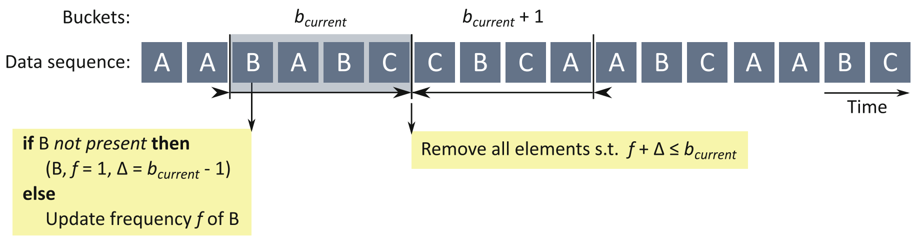

Buckets:
$b_{current}$
$b_{current} + 1$
Data sequence:
A A B A B C C B C A A B C A A B C
Time
if B *not present* then
(B, $f = 1$, $\Delta = b_{current} - 1$)
else
Update frequency f of B
Remove all elements s.t. $f + \Delta \leq b_{current}$

Fig. 3. Graphical representation of the Lossy Counting algorithm.

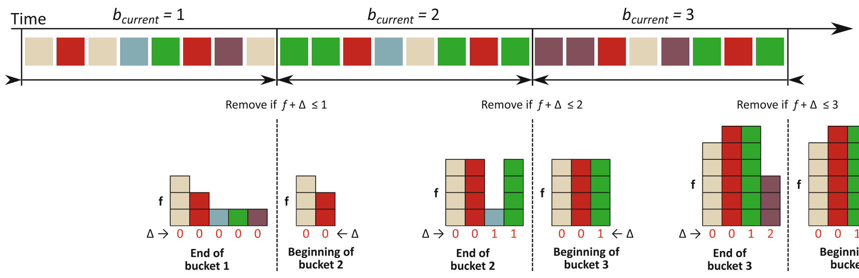


Fig. 4. Demonstration of the evolution of the internal data structure constructed by the Lossy Counting on a simple stream. Each color refers to a variable.

of its data structure T depends on the stream and on the max approximation error (i.e., if the error is set to 0 and the observations never repeat, the size of T will grow indefinitely). Variants of the algorithm enforcing a fixed memory bound is available as well [18] but are not described in detail here. In the rest of this chapter, a set whose entries are structured and updated using the Lossy Counting algorithm (i.e., T in Algorithm 2) will be called Lossy Counting Set.

Figure 4 shows a demonstration of the evolution of the Lossy Counting Sets over time (at the end and at the beginning of each virtual bucket) for the given stream. In this case, for simplicity purposes, the background color of each box represents the variable that we are counting. The counting is represented as the stacking of the blocks, and below, in red, the maximum error for each variable is ported.

The most relevant benefit of reducing the problem to a known one is the ability to employ existing as well as new solutions in order to improve the efficiency of the final process mining solution. Clearly, this desirable feature is paid back in terms of the complexity of the steps (both conceptual and computational) required for the translation.

Offline Computation. Due to some of the constraints imposed by the streaming paradigm, one option consists of moving parts of the computation offline (cf. Fig. 2) so that performance requirements are met when the system goes online. This idea implies decomposing the problem into sub-problems and reflecting on whether some of them can be solved without the actual streaming data. If that

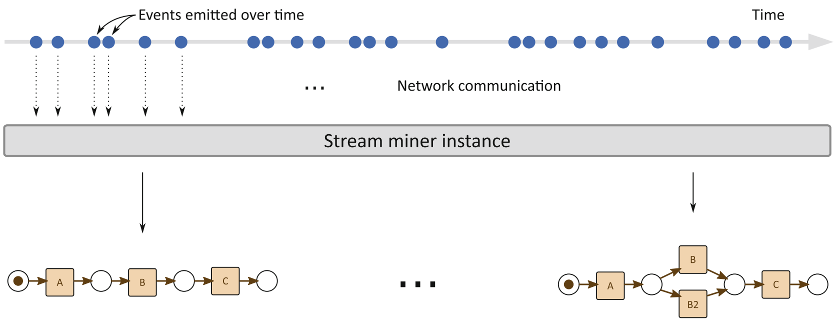


Fig. 5. Conceptualization of the streaming process discovery. Figure from [16].

is the case, these sub-problems will be solved beforehand and corresponding pre-computed solutions will be available as the events are coming in.

Such an approach comes with the advantage of caching the results of computations that would otherwise require extremely expensive computations. This approach, still, suffers from several limitations since it is not possible to apply this approach to all streaming process mining problems. Additionally, by computing everything in advance, we lose the possibility of adapting the pre-computed solutions to the actual context, which might be uniquely specific to the running process instance.

Hybrid Approaches. As a final note, we should not rule out the option of defining *ensemble* methods that combine different approaches together (see Fig. 2).

3 Streaming Process Discovery

After introducing the general principles and taxonomy of techniques to tackle streaming process mining, in this section, we will specifically analyze the problem of streaming process discovery.

A graphical conceptualization of the problem is reported in Fig. 5: the basic idea is to have a source of events that generates an event stream. Such an event stream is consumed by a miner which keeps a representation of the underlying process model updated as the new events are coming in.

3.1 State of the Art

In this section, the main milestones of the streaming process discovery will be presented. The first available approach to tackle the streaming process discovery problem is reported in [15,16]. This technique employs a "problem reduction" approach (cf. Fig. 2) rephrasing the Heuristics Miner [38] as a frequency counting problem. The details of this approach will be presented in Sect. 3.2. In [23], authors present StrProM which, similarly to the previous case, tracks the direct following relationship by keeping a prefix tree updated with Lossy Counting with Budget [18].

More recently, an architecture called S-BAR, which keeps an updated abstract representation of the stream (e.g., direct follow relationships), is used as starting point to infer an actual process model, as described in [44]. Different algorithms (including α. [39], Heuristics Miner [38] and Inductive Miner [26]) have been incorporated to be used with this approach. Also in this case authors reduced their problem to frequency counting, thus using Lossy Counting, Space Saving [32], and Frequent [24].

Declarative processes (cf. Chapter 4) have also been investigated as the target of the discovery. In [12,14,27], authors used the notion of "replayers" – one for each Declare [35] template to mine – to discover which one are fulfilled. Also in this case, Lossy Counting strategies have been employed to achieve the goal. A newer version of the approach [33], is also capable of discovering data conditions associated with the different constraints.

3.2 Heuristics Miner with Lossy Counting (HM-LC)

This section describes in more detail one algorithm for streaming process discovery: Heuristics Miner with Lossy Counting (HM-LC) [16].

The Heuristics Miner algorithm [38] is a discovery algorithm which, given the frequency of the direct following relations observed (reported as $|a > b|$ and indicating the number of times b is observed directly after a), calculates the *dependency measure*, a measure of the strength of the causal relation between two activities a and b:

$$a \Rightarrow b = \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1} \in [-1, 1]. \tag{1}$$

The closer the value of such metric is to 1, the stronger the causal dependency from a to b. Based on a given threshold (parameter asked as input), the algorithm considers only those dependencies with values exceeding the threshold, deeming the remaining as noise. By considering all dependencies which are strong enough, it is possible to build a dependency graph, considering one node per activity and one edge for each dependency. In such a graph, however, when splits or joins are observed (i.e., activities with more than one outgoing or incoming connection) it is not possible to distinguish the type of the splits. In the case of a dependency from a to b and also from a to c, Heuristics Miner disambiguates between an AND and an XOR split by calculating the following metric (also based on the frequency of direct following relations):

$$a \Rightarrow (b \wedge c) = \frac{|b > c| + |c > b|}{|a > b| + |a > c| + 1} \in [0, 1]. \tag{2}$$

When the value of this measure is high (i.e. close to 1), it is likely that b and c can be executed in parallel, otherwise, these will be mutually exclusive. As for the previous case, a threshold (parameter asked as input) is used to make the distinction.

It is important to note that the two fundamental measures employed by the Heuristics Miner rely on the frequency of the directly-follows measure (e.g. $|a > b|$), and so the basic idea of Heuristics Miner with Lossy Counting is to consider such values as "variables" to be observed in a stream, thus reducing the problem to frequency counting.

As previously mentioned, Lossy Counting is an algorithm for frequency counting. In particular, the estimated frequencies can be characterized both in terms of lower and upper bounds as follows: given the estimated frequency f (i.e., the frequency calculated by the algorithm), the true frequency F (i.e., the actual frequency of the observed variable), the maximum approximation error ϵ and the number of events observed N, these inequalities hold

$$f \leq F \leq f + \epsilon N.$$

To calculate the frequencies, Lossy Counting uses a set data structure where each element refers to the variable being counted, its current (and approximated) frequency, and the maximum approximation error in the counting of that variable.

For the sake of simplicity, in Heuristics Miner with Lossy Counting, the observable units of the event stream comprises just the activity name and the case id (cf. Definition 1). In other words, each event observed from the stream comprises two attributes: the activity name and the case id (cf. Definition 1 [1], Sect. 3.2 [1]), so an event e is a tuple with $e = (c, a)$, where $\#_{case}(e) = c$ and $\#_{act}(e) = a$.

The pseudocode of HM-LC is reported in Algorithm 3. The fundamental idea of the approach is to count the frequency of the direct following relations observed. In order to achieve this goal, however, it is necessary to *identify* the direct following pairs in the first place. As depicted in Fig. 6, to identify direct following relations it is first necessary to disentangle the different traces that are intertwined in an event stream. To this end, the HM-LC instantiates two Lossy Counting Sets: $\mathcal{D}_C$, and $\mathcal{D}_R$. The first keeps track of the latest activity observed in each process instance whereas the second counts the actual frequency of the directly follow relations. These data structures are initialized at the first line of the algorithm, which is followed by the initialization of the counter of observed events (line 2) and the calculation of the size of the buckets (line 3). After the initial setup of the data structure, a never-ending loop starts by observing events from the stream (line 5, cf. Definition 1), where each

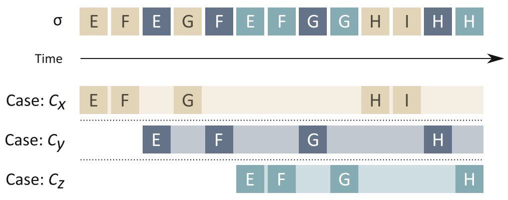


Fig. 6. Conceptualization of the need to isolate different traces based on a single stream. Boxes represent events: their background colors represent the case id, and the letters inside are the activity names. First line reports the stream, following lines are the single cases. Figure from [16].

Algorithm 3: Heuristics Miner with Lossy Counting (simplified)

```
   Input: $S$: event stream
          $\epsilon$: approximation error
1  Initialize Lossy Counting Sets $\mathcal{D}_C$ and $\mathcal{D}_R$
2  $N \leftarrow 1$                                         // Counter of observed events
3  $w \leftarrow \lceil \frac{1}{\epsilon} \rceil$                                                   // Bucket size
4  forever do
5      $(c_N, a_N) \leftarrow observe(S)$
6      $b_{curr} = \lceil \frac{N}{w} \rceil$                        // Calculate the current bucket id
       // Step 1: Update the Lossy Counting Sets
7      if $\exists((c, a_{last}), f, \Delta) \in \mathcal{D}_C$ such that $c = c_N$ then
8          Remove the entry $((c, a_{\text{last}}), f, \Delta)$ from $\mathcal{D}_C$
9          $\mathcal{D}_C \leftarrow \mathcal{D}_C \cup \{((c, a_N), f + 1, \Delta)\}$
           // Update the $\mathcal{D}_R$ data structure
10         $r_N \leftarrow (a_{\text{last}}, a_N)$ // Build relation $r_N$ as $a_{\text{last}} \rightarrow a_N$
11         if $\exists(r, f, \Delta) \in \mathcal{D}_R$ such that $r = r_N$ then
12             Remove the entry $(r, f, \Delta)$ from $\mathcal{D}_R$
13             $\mathcal{D}_R \leftarrow \mathcal{D}_R \cup \{(r, f + 1, \Delta)\}$
14         else
15             $\mathcal{D}_R \leftarrow \mathcal{D}_R \cup \{(r_N, 1, b_{curr} - 1)\}$
16         end
17     else
18         $\mathcal{D}_C \leftarrow \mathcal{D}_C \cup \{((c_N, a_N), 1, b_{curr} - 1)\}$
19     end
       // Step 2: Periodic cleanup
20     if $N \equiv 0 \mod w$ then
21         forall the $((c, a), f, \Delta) \in \mathcal{D}_C$ such that $f + \Delta \leq b_{curr}$ do
22             Remove $((c, a), f, \Delta)$ from $\mathcal{D}_C$
23         end
24         forall the $(r, f, \Delta) \in \mathcal{D}_R$ such that $f + \Delta \leq b_{curr}$ do
25             Remove $(r, f, \Delta)$ from $\mathcal{D}_R$
26         end
27     end
28     $N \leftarrow N + 1$
       // Step 3: Consumption of the data structure to update the model
29     Update the model using $\mathcal{D}_R$
30 end
```

event is the pair (c_N, a_N), indicating that the case id observed as event N is c_N (resp., the activity is a_N). The id of the current bucket is calculated right afterwards (line 6). The whole algorithm is then divided into three conceptual steps: in the first the data structures are updated; in the second periodic cleanup takes place; in the third the data structures are used to construct and update the actual model.

Step 1: Updating the Data Structure. The Lossy Counting Set $\mathcal{D}_C$ has been defined in order not only to keep a count of the frequency of each case id observed in the events but also to keep track of the latest activity observed in the given trace. To achieve this goal, the entries of the data structure are tuples themselves, comprising the case id as well as the name of the latest activity observed in the case. Therefore, the first operation within the step consists of checking for the presence of an entry in $\mathcal{D}_C$ matching the case id of the observed event (but not the activity name), as reported in line 7. If this is the case, the data structure $\mathcal{D}_C$ is updated, not only by updating the frequency but also by updating the latest activity observed in the given case (lines 8 and 9). In addition, having already an entry in $\mathcal{D}_C$ means that a previous event within the same trace has already been seen and, therefore, it is possible to construct a direct following relation (cf. line 10 of Algorithm 3). This relation is then treated as a normal variable to be counted and the corresponding Lossy Counting Set is updated accordingly (lines 11–16). In case $\mathcal{D}_C$ did not contain an entry referring to case id c_N, it means that the observed event is the first event of its process instance (up to the approximation error) and hence just a new entry in $\mathcal{D}_C$ is inserted and no direct following relation is involved (line 18).

Step 2: Periodic Cleanup. With a periodicity imposed by the maximum approximation error (ϵ), i.e., at the end of each bucket (line 20), the two Lossy Counting Sets are updated by removing entries that are not frequent or recent enough (lines 21–26). Please note that the algorithm expects that observing an event belonging to a process instance that has been removed from $\mathcal{D}_C$ corresponds to losing one direct following relation from the counting in $\mathcal{D}_R$. From this point of view, the error on the counting of the relations is not only affected by $\mathcal{D}_R$ but, indirectly, also by the removal of instances from $\mathcal{D}_C$ which causes a relation not to be seen at all (and therefore, it cannot be counted).

Step 3: Consumption of the Data Structures. The very final step of the algorithm (line 29) consists of triggering a periodic update of the model. The update procedure (not specified in the algorithm) extracts all the activities involved in a direct following relations from $\mathcal{D}_R$ and uses the dependency measure (cf. Eq. 1) to build a dependency graph, by keeping the relations with dependency measure above a threshold. To disambiguate AND/XOR splits Eq. 2 is used. Both these measures need to be adapted in order to retrieve the frequency of the relations from $\mathcal{D}_R$.

The procedure just mentioned recomputes the whole model from scratch. However, observing a new event will cause only local changes to a model. Hence a complete computation of the whole model is not necessary. In particular, it is possible to rearrange Eqs. 1 and 2 in order to signal when a dependency has changed. Specifically, given a dependency threshold τ_{dep}, we know that a dependency should be present if these inequalities hold:

$$|a > b| \geq \frac{|b > a|(1 + \tau_{dep}) + \tau_{dep}}{1 - \tau_{dep}} \quad \text{or} \quad |b > a| \leq \frac{|a > b|(1 - \tau_{dep}) - \tau_{dep}}{1 + \tau_{dep}}$$

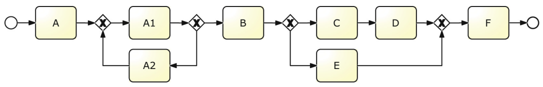


Fig. 7. Reference process model used to calculate the conformance of traces in Table 1, from [17].

Table 1. Example traces with corresponding offline conformance.

Trace	Conf.
$t_1 = \langle A, A1, B, E, F \rangle$	1.00
$t_2 = \langle A, A1, A2, A1, B \rangle$	0.80
$t_3 = \langle B, C, D, F \rangle$	0.78
$t_4 = \langle B, C, D \rangle$	0.62

In a similar fashion, we can rewrite Eq. 2 so that, given an AND threshold parameter τ_{and}, a split (i.e., from activity a to both activities b and c) has type AND if all these inequalities hold:

$$\begin{aligned}
|b > c| &\leq \tau_{and}\left(|a > b| + |a > c| + 1\right) - |c > b| \\
|c > b| &\leq \tau_{and}(|a > b| + |a > c| + 1) - |b > c| \\
|a > b| &\leq \frac{|b > c| + |c > b|}{\tau_{and}} - |a > c| - 1 \\
|a > c| &\leq \frac{|b > c| + |c > b|}{\tau_{and}} - |a > b| - 1
\end{aligned}$$

If this is not the case, the type of the split will be XOR. Therefore, by monitoring how the frequencies of some of the relations in $\mathcal{D}_R$ (which should be used as an approximation of the direct following frequencies) are evolving, it is possible to pinpoint the changes appearing in a model, with no need for rebuilding it from scratch all the times.

In this section, we did not exhaustively cover the reduction of the Heuristics Miner to Lossy Counting (for example, we did not consider the absolute number of observations for an activity or parameters such as the relative-to-best) but we focused on the core aspects of the reduction. The goal of the section was to present the core ideas behind a streaming process discovery algorithm while, at the same time, showing an example of an algorithm based on the problem reduction approach (cf. Fig. 2).

4 Streaming Conformance Checking

Computing the conformity of running instances starting from events observed in a stream is the main goal of streaming conformance checking.

Consider, for example, the process model reported in Fig. 7 as a reference process, and let's investigate the offline conformance (calculated according to the alignment technique reported in [2]) for the traces reported in Table 1. Trace t_1 is indeed conforming with respect to the model as it represents a possible complete execution of the process. This information is already properly captured by the offline analysis. Trace t_2, on the other hand, is compliant with the process but just up to activity B, as reported by the conformance value 0.8:

offline systems assume that the executions are complete, and therefore observing an incomplete trace represents a problem. However, as previously discussed and as shown in Fig. 1a, in online settings it could happen that parts of the executions are missing due to the fact that the execution has not yet arrived at this part of the computation. This *could* be the case with trace t_2 (i.e., t_2 is the prefix of a compliant trace). Trace t_3 suffers from the opposite problem: the execution is conforming to the model, but just from activity B onward. While offline conformance, in this case, is calculated to the value of 0.68, as for the previous case, we cannot rule out the option that the trace is actually compliant but, since the trace started before the streaming conformance checker was online, it was incapable of analyzing the beginning of it (i.e., t_3 is the suffix of a compliant trace). Trace t_4, finally, seems compliant just between activities B and D. Though offline conformance, in this case, is 0.62, as for the previous two cases, in a streaming setting, we cannot exclude that the issue actually derives from the combination of the trace starting before the streaming conformance was online and the trace not being complete (i.e.,t_4 is a subsequence of a compliant trace).

Hopefully, discussing the previous examples helped to point out the limit of calculating the extent of the conformance using only one numerical value in a streaming setting. Indeed, when the assumption that executions are complete is dropped, the behavior shown in the traces of Table 1 could become 100% compliant since the actual issue does not lie in the conformity but in the amount of observed behavior.

4.1 State of the Art

Computing the conformity of a stream with respect to a reference model has received a fairly large amount of attention, in particular in the case of declarative processes. Under the name "*operational support*", research has been focusing [28, 29] on understanding if and which constraints are violated and satisfied as new events are coming in. In particular, each constraint is associated with one of four possible truth values: permanently or temporarily violated or fulfilled which are computed by representing the behavior as an automaton with all executions replayed on top of it.

Streaming conformance checking on imperative models has also received attention, though more recently. Optimal alignments can be computed for the prefix (i.e., prefix-alignments) of the trace seen up to a given point in time [41], resulting in a very reliable approach which, however, meets only to some extent the streaming scenario (cf. Sect. 1.2). A more recent approach [36] is capable of improving the performance of calculating a prefix-alignment, by rephrasing the problem as the shortest path one and by incrementally expanding the search space and reusing previously computed intermediate results.

A different line of research focused on calculating streaming conformance for all scenarios (cf., Fig. 1a). In this case, techniques employed "offline computation" approaches [11,17,25] to construct data structures capable of simplifying the computation when the system goes online. These approaches not only compute the conformity of a running instance but also try to quantify the amount of behavior observed or still to come.

In addition to these, one of the first approaches [45] focused on a RESTful service capable of performing the token replay on a BPMN model (via a token pull mechanism). No explicit guarantees, however, are reported concerning the memory usage, the computational complexity, or the reliability of the results, suggesting that the effort was mostly on the interface type (i.e., *online* as in *RESTful*).

4.2 Conformance Checking with Behavioral Patterns

This section presents in more detail one algorithm for streaming conformance checking using behavioral patterns [17]. The algorithm belongs to the category of offline computation (cf. Fig. 2), where the heaviest computation is moved before the system goes online, thus meeting the performance requirement of streaming settings.

The fundamental idea of the approach is that using just one metric to express conformity could lead to misleading results, i.e. cases that already started and/or that are not yet finished get falsely penalized. To solve these issues, the approach proposes to break the conformity into three values:

1. *Conformance*: indicating the amount of actually correct behavior seen;
2. *Completeness*: providing an estimate of the extent to which the trace has been observed since the beginning; and
3. *Confidence*: indicating how much of the trace has been seen, and therefore to what extent the conformance is likely to remain stable.

A graphical representation of these concepts is reported in Fig. 8. In addition, the approach does not assume any specific modeling language for the reference process. Instead, the approach takes the reference process as a constraining of the relative orders of its activities. Such constraints are defined in terms of behavioral patterns, such as weak ordering, parallelism, causality, and conflict. Such behavioral patterns (with the corresponding activities involved) represent also what the conformance checking algorithm observes. In the context of this chapter, we will consider the directly follow relation as a pattern.

Please note that the input of the algorithm is not a stream of events, but a stream of observed behavioral patterns, which could require some processing of the raw events. This, however, does not represent a problem for the behavioral pattern considered (i.e., directly follow relation), since these can be extracted using the technique described in Sect. 3.2.

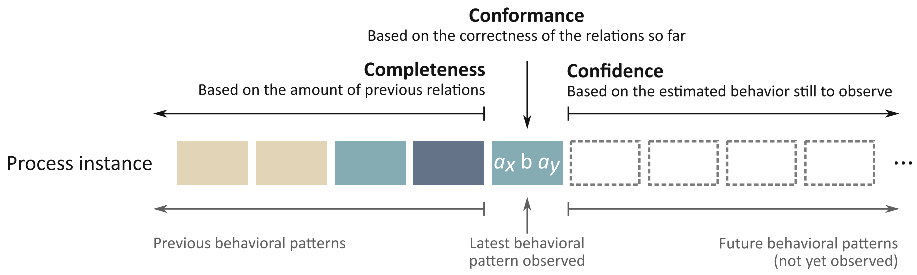

Conformance
Based on the correctness of the relations so far
Completeness
Based on the amount of previous relations
Confidence
Based on the estimated behavior still to observe
Process instance
a_x b a_y
...
Previous behavioral patterns
Latest behavioral pattern observed
Future behavioral patterns (not yet observed)

Fig. 8. General idea of the 3 conformance measures computed based on a partially observed process instance: *conformance*, *completeness*, and *confidence*. Figure from [17].

As previously mentioned, the technique offloads the computation to a preprocessing stage which takes place offline, before the actual conformance is computed. During such a step, the model is converted into another representation, better suited for the online phase. Specifically, the new model contains:

1. The set of behavioral patterns that the original process prescribes;
2. For each of the behavioral patterns identified, the minimum and maximum number of distinct prescribed patterns that must occur *before* it, since the very beginning of the trace;
3. For each behavioral pattern, the minimum number of distinct patterns *still to observe* to reach a valid accepting state of the process (as prescribed by the reference model).

These requirements drive the definition of the formal representation called "Process Model for Online Conformance" (PMOC). A *process model for online conformance* $M = (B, P, F)$ is defined as a triplet containing the set of prescribed behavioural patterns B. Each pattern $b(a_1, a_2)$ is defined as a relation b (e.g., the directly follow relation) between activities $a_1, a_2 \in \mathcal{U}_{act}$ (cf. Definition 1 [1]). P contains, for each behavioral pattern $b \in B$, the pair of minimum and maximum number distinct prescribed patterns (i.e., B) to be seen before b. We refer to these values as $P_{\min}(b)$ and $P_{\max}(b)$. For each pattern, $b \in B$, $F(b)$ refers to the minimum number of distinct patterns (i.e., B) required to reach the end of the process from b.

Once such a model is available, the conformance values can be calculated according to Algorithm 4 which executes three steps for each event: updating the data structures, calculating the conformance values, and housekeeping cleanup. After two maps are initialized (lines 1, 2), the never-ending loop starts and, each observation from the stream (which refers to a behavioral patter b for case id c, cf. Definition 1) triggers and update of the two maps: if the pattern refers to a prescribed relation, then it is added to the $\mathsf{obs}(c)$ set (line 6)[4], otherwise, the value of incorrect observations for the process instance $\mathsf{obs}(c)$ is incremented (line 8)[5]. In the second step, the algorithm calculates the new conformance values.

[4] If obs has no key c, $\mathsf{obs}(c)$ returns the empty set.
[5] If inc has no key c, then $\mathsf{inc}(c)$ returns 0.

Algorithm 4: Conformance Checking with Behavioral Patterns

```
   Input: $S$: stream of behavioural patterns
          $M = (B, P, F)$: process model for online conformance
 1 Init map obs         // Maps case ids to set of observed patterns from $M$
 2 Init map inc                                  // Maps case ids to integers
 3 forever do
 4     $(c, b) \leftarrow observe(S)$        // New observation of pattern $b$ for case $c$
       // Step 1: update internal data structures
 5     if $b \in B$ then
 6         $\mathsf{obs}(c) \leftarrow \mathsf{obs}(c) \cup \{b\}$     // If $b$ already in $\mathsf{obs}(c)$, then no effect
 7     else
 8         $\mathsf{inc}(c) \leftarrow \mathsf{inc}(c) + 1$
 9     end
       // Step 2: compute online conformance values
10     $\mathsf{conformance}(c) \leftarrow \frac{|\mathsf{obs}(c)|}{|\mathsf{obs}(c)| + \mathsf{inc}(c)}$
11     Notify new value of $\mathsf{conformance}(c)$
12     if $b \in B$ then
13         if $P_{\min}(b) \le |obs(c)| \le P_{\max}(b)$ then
14             $\mathsf{completeness}(c) \leftarrow 1$
15         else
16             $\mathsf{completeness}(c) \leftarrow \min\left\{1, \frac{|\mathsf{obs}(c)|}{P_{\min}(b) + 1}\right\}$
17         end
18         $\mathsf{confidence}(c) \leftarrow 1 - \frac{F(b)}{\max_{b' \in B} F(b')}$
19         Notify new values of $\mathsf{completeness}(c)$ and $\mathsf{confidence}(c)$
20     end
       // Step 3: cleanup
21     if size of obs and inc is close to max capacity then
22         Remove oldest entries from obs and inc
23     end
24 end
```

The actual conformance, which resembles the concept of *precision*, is calculated (lines 10, 11) as the number of distinct observed prescribed patterns in c (i.e., $|\mathsf{obs}(c)|$) divided by the sum of the number of prescribed observed patterns and the incorrect patterns (i.e., $|\mathsf{obs}(c)| + \mathsf{inc}(c)$): 1 indicates full conformance (i.e., only correct behaviour) and 0 indicates no conformance at all (i.e., only incorrect behaviour). Completeness and confidence are updated only when a prescribed behavioral pattern is observed (line 12) since they require locating the pattern itself in the process. Concerning completeness, we have perfect value if the number of distinct behavioral patterns observed so far is within the expected interval for the current pattern (lines 13, 14). If this is not the case, we might have seen fewer or more patterns than expected. If we have seen fewer patterns, the completeness is the ratio of observed patterns over the minimum expected; otherwise, it's just 1 (i.e., we observed more patterns than needed, so the completeness is not an issue). Please bear in mind that these numbers confront the *number* of distinct patterns, not their type, thus potentially leading to false positives (line 16). The confidence is calculated (line 18) as 1 minus the proportion of patterns to observe (i.e., $F(b)$) and the overall maximum number of future patterns (i.e., $\max_{b' \in B} F(b')$): a confidence level 1 indicates strong confidence (i.e., the execution reached the end of the process), 0 means low confidence (i.e., the execution is still far from completion, therefore there is room for change). The final step performs some cleanup operations on obs and inc (lines 21–23). The algorithm does not specify how old entries should be identified and removed, but, as seen on the previous section, existing approaches can easily handle this problem (e.g., by using a Lossy Counting Set).

It is important to note once again that the actual algorithm relies on a data structure (the PMOC) that is tailored to the purpose and that might be computational very expensive to obtain. However, since this operation is done only once and before any streaming processing, this represents a viable solution. The details on the construction of the PMOC are not analyzed in detail here but are available in [17]. Briefly, considering the directly follow relation as the behavioral pattern, the idea is to start from a Petri net and calculate its reverse (i.e., the Petri net where all edges have opposite directions). Both these models are then unfolded according to a specific stop criterion and, once corresponding reachability graphs are computed, the PMOC can be easily derived from the reachability graph of the unfolded original Petri net and the reachability graph of the unfolded reverse net.

Considering again the traces reported in Table 1 and the reference model in Fig. 7, all traces have a streaming conformance value of 1 (when calculated using the approach just described). The completeness is 1 for t_1 and t_2, 0.6 for t_3, and 0.5 for t_4. The confidence is 1 for t_1 and t_3, 0.5 for t_2, and 0.75 for t_4. These values indeed capture the goals mentioned at the beginning of this section: do not penalize the conformance but highlight the actual issues concerning the missing beginning or end of the trace.

As for the streaming process discovery case, in this section, we did not exhaustively cover the algorithm for streaming conformance checking presented. Instead, we focused on the most important aspects of the approach, hopefully also giving an intuition of how an offline computation approach could work (cf. Fig. 2).

5 Other Applications and Outlook

It is worth mentioning that the concepts related to streaming process mining have been applied not only to the problem of discovery and conformance but, to a limited extent, to other challenges.

Examples of such applications are the discovery of cooperative structures out of event streams, as tackled in [43], where authors process an event stream and update the set of relationships of a cooperative resource network. In [40], several additional aspects of online process mining are investigated too.

Supporting the research in streaming process mining has also been a topic of research. Simulation techniques have been defined both as standalone applications [9], as ProM plugins [42], or just as communication protocols [13].

Finally, from the industrial point of view, it might be interesting to observe that while some companies are starting to consider some aspects related to the topics discussed in this chapter (e.g., Celonis' Execution Management Platform supports the real-time data ingestion, though not the analysis), none of them offers actual solutions for streaming process mining. A report from Everest Group[6] explicitly refers to real-time monitoring of processes as an important process intelligence capability not yet commercially available.

This chapter presented the topic of streaming process mining. While the field is relatively young, several techniques are already available both for discovery and conformance checking.

We presented a taxonomy of the existing approaches which, hopefully, can be used proactively, when new algorithms need to be constructed, to identify how a problem can be tackled. Then two approaches, one for control-flow discovery and one for conformance checking, are presented in detail which, in addition, belong to different categories of the taxonomy. Alongside these two approaches, window models can also be employed, yet their efficacy is typically extremely low compared to algorithms specifically designed for the streaming context.

It is important to mention that streaming process mining has very important challenges still to be solved. For example, dealing with a stream where the arrival time of events does not coincide with their actual execution. In this case, it would be necessary to reorder the list of events belonging to the same process instance before processing them. Another relevant issue might be the inference of the termination of process instances. Finally, so far, we always considered an insert-only stream model, where events can only be added in a monotonic fashion. Scenarios where observed events can be changed or removed (i.e., insert-delete models) are yet to be considered.

References

1. van der Aalst, W.M.P.: Chapter 1 - Process mining: a 360 degrees overview. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. Lecture Notes in Business Information Processing, pp. ??-??, vol. 448. Springer-Verlag, Berlin (2022)
2. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Technische Universiteit Eindhoven (2014)
3. Aggarwal, C.C.: Data Streams: Models and Algorithms. Advances in Database Systems. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-47534-9
4. Berry, R.F., McKenney, P.E., Parr, F.N.: Responsive systems: an introduction. IBM Syst. J. **47**(2), 197–206 (2008)

[6] https://www2.everestgrp.com/reportaction/EGR-2020-38-R-3808/Marketing.

5. Beverungen, D., et al.: Seven paradoxes of business process management in a hyperconnected world. Bus. Inf. Syst. Eng. **63**(2), 145–156 (2021)
6. Bifet, A., Gavaldà, R., Holmes, G., Pfahringer, B.: Machine Learning for Data Streams. The MIT Press, Cambridge (2018)
7. Bifet, A., Kirkby, R.: Data stream mining: a practical approach. Technical report, Centre for Open Software Innovation - The University of Waikato (2009)
8. Burattin, A.: Process Mining Techniques in Business Environments. Lecture Notes in Business Information Processing, vol. 207. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-17482-2
9. Burattin, A.: PLG2: multiperspective process randomization with online and offline simulations. In: Online Proceedings of the BPM Demo Track (2016). CEUR-WS.org
10. Burattin, A.: Streaming process discovery and conformance checking. In: Sakr, S., Zomaya, A., (eds.) Encyclopedia of Big Data Technologies. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63962-8_103-1
11. Burattin, A., Carmona, J.: A framework for online conformance checking. In: Teniente, E., Weidlich, M. (eds.) BPM 2017. LNBIP, vol. 308, pp. 165–177. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74030-0_12
12. Burattin, A., Cimitile, M., Maggi, F.M., Sperduti, A.: Online discovery of declarative process models from event streams. IEEE Trans. Serv. Comput. **8**(6), 833–846 (2015)
13. Burattin, A., Eigenmann, M., Seiger, R., Weber, B.: MQTT-XES: real-time telemetry for process event data. In: CEUR Workshop Proceedings (2020)
14. Burattin, A., Maggi, F.M., Cimitile, M.: Lights, camera, action! Business process movies for online process discovery. In: Proceedings of the 3rd International Workshop on Theory and Applications of Process Visualization (TAProViz 2014) (2014)
15. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Heuristics Miners for Streaming Event Data. ArXiv CoRR, December 2012
16. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: Proceedings of the IEEE Congress on Evolutionary Computation, pp. 2420–2427. IEEE (2014)
17. Burattin, A., van Zelst, S.J., Armas-Cervantes, A., van Dongen, B.F., Carmona, J.: Online conformance checking using behavioural patterns. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 250–267. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_15
18. Da San Martino, G., Navarin, N., Sperduti, A.: A lossy counting based approach for learning on streams of graphs on a budget. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, pp. 1294–1301. AAAI Press (2012)
19. Dayarathna, M., Perera, S.: Recent advancements in event processing. ACM Comput. Surv. **51**(2), 1–36 (2018)
20. Gaber, M.M., Zaslavsky, A., Krishnaswamy, S.: Mining data streams: a review. ACM SIGMOD Rec. **34**(2), 18–26 (2005)
21. Gama, J.: Knowledge Discovery from Data Streams. Chapman and Hall/CRC, London (2010)
22. Gandomi, A., Haider, M.: Beyond the hype: big data concepts, methods, and analytics. Int. J. Inf. Manage. **35**(2), 137–144 (2015)
23. Hassani, M., Siccha, S., Richter, F., Seidl, T.: Efficient process discovery from event streams using sequential pattern mining. In: 2015 IEEE Symposium Series on Computational Intelligence, pp. 1366–1373 (2015)

24. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. **28**(1), 51–55 (2003)
25. Jonathan Lee, W.L., Burattin, A., Munoz-Gama, J., Sepúlveda, M.: Orientation and conformance: a HMM-based approach to online conformance checking. Inf. Syst. **102**, 1–38 (2020)
26. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_17
27. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 433–448. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38709-8_28
28. Maggi, F.M., Montali, M., van der Aalst, W.M.P.: An operational decision support framework for monitoring business constraints. In: Proceedings of 15th International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 146–162 (2012)
29. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_13
30. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of International Conference on Very Large Data Bases, pp. 346–357. Morgan Kaufmann, Hong Kong, China (2002)
31. Mans, R., van der Aalst, W.M.P., Vanwersch, R.J.B.: Process Mining in Healthcare. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16071-9
32. Metwally, A., Agrawal, D., El Abbadi, A.: Efficient computation of frequent and top-k elements in data streams. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 398–412. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30570-5_27
33. Navarin, N., Cambiaso, M., Burattin, A., Maggi, F.M., Oneto, L., Sperduti, A.: Towards online discovery of data-aware declarative process models from event streams. In: Proceedings of the International Joint Conference on Neural Networks (2020)
34. Patroumpas, K., Sellis, T.: Window specification over data streams. In: Proceedings of Current Trends in Database Technology - EDBT, pp. 445–464 (2006)
35. Pešić, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: Proceedings of EDOC, pp. 287–298. IEEE (2007)
36. Schuster, D., van Zelst, S.J.: Online process monitoring using incremental state-space expansion: an exact algorithm. In: Fahland, D., Ghidini, C., Becker, J., Dumas, M. (eds.) BPM 2020. LNCS, vol. 12168, pp. 147–164. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58666-9_9
37. Sharp, A.M.: Incremental Algorithms: Solving Problems in a Changing World. Ph.D. thesis, Cornell University (2007)
38. van der Aalst, W.M.P., Ton, A.J., Weijters, M.M.: Rediscovering workflow models from event-based data using little thumb. Integr. Comput. Aid. Eng. **10**(2), 151–162 (2003)

39. van der Aalst, W.M.P., Ton, A.J., Weijters, M.M., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**, 1128–1142 (2004)
40. van Zelst, S.J.: Process mining with streaming data. Ph.D. thesis, Technische Universiteit Eindhoven (2019)
41. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B., van der Aalst, W.M.P.: Online conformance checking: relating event streams to process models using prefix-alignments. Int. J. Data Sci. Anal. **8**, 269–284 (2017)
42. van Zelst, S.J., van Dongen, B., van der Aalst, W.M.P.: Know what you stream: generating event streams from CPN models in ProM 6. In: CEUR Workshop Proceedings, pp. 85–89 (2015)
43. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: Online discovery of cooperative structures in business processes. In: Debruyne, C., et al. (eds.) OTM 2016. LNCS, vol. 10033, pp. 210–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48472-3_12
44. van Zelst, S.J., van Dongen, B., van der Aalst, W.M.P.: Event stream-based process discovery using abstract representations. Knowl. Inf. Syst. **54**, 1–29 (2018)
45. Weber, I., Rogge-Solti, A., Li, C., Mendling, J.: CCaaS: online conformance checking as a service. In: Proceedings of the BPM Demo Session 2015, vol. 1418, pp. 45–49 (2015)
46. Widmer, G., Kubat, M.: Learning in the presence of concept drift and hidden contexts. Mach. Learn. **23**(1), 69–101 (1996)