Recurrent Process Mining on Procedural and Declarative Approaches

Alifah Syamsiyah, Boudewijn F. van Dongen, Wil M.P. van der Aalst

Eindhoven University of Technology
A.Syamsiyah@tue.nl, B.F.v.Dongen@tue.nl, W.M.P.v.d.Aalst@tue.nl

Abstract. In organizations, process mining activities are typically performed in a recurrent fashion, e.g. once a week, an event log is extracted from the information systems and a process mining tool is used to analyze the process' characteristics. Typically, process mining tools import the data from a file-based source in a pre-processing step, followed by an actual process discovery step over the pre-processed data in order to present results to the analyst. As the amount of event data grows over time, these tools take more and more time to do pre-processing and all this time, the business analyst has to wait for the tool to finish. In this paper, we consider the problem of recurrent process discovery in live environments, i.e. in environments where event data can be extracted from information systems near real time. We present a method that pre-processes each event when it is being generated, so that the business analyst has the pre-processed data at his/her disposal when starting the analysis. To this end, we define a notion of intermediate structure between the underlying data and the layer where the actual mining is performed. This intermediate structure is kept in a persistent storage and is kept live under updates. Using two state-of-the-art process mining techniques, we show the feasibility of our approach. Our work is implemented in process mining tool ProM using a relational database system as our persistent storage. Experiments are presented on real-life event data to compare the performance of the proposed approach with the state of the art.

Keywords: recurrent proses mining, live event data, incremental process discovery

1 Introduction

Process mining is a discipline where the aim is to improve an organization's processes given the information from the so called *event logs* [17]. Process mining techniques have been successfully demonstrated in various case studies such as health care, insurance, and finance [5,13,15,24]. In many of these cases, a one-time study was performed, but in practice, process mining is typically a recurring activity, i.e. an activity that is performed on a routine basis.

As an illustration, suppose that each manager of an insurance company has the obligation to report her/his work to a director once in each month to see the company's progress since the beginning of a year. Typical analysis in such regular report incorporates the observations from previous month, last three months, or last year. This type

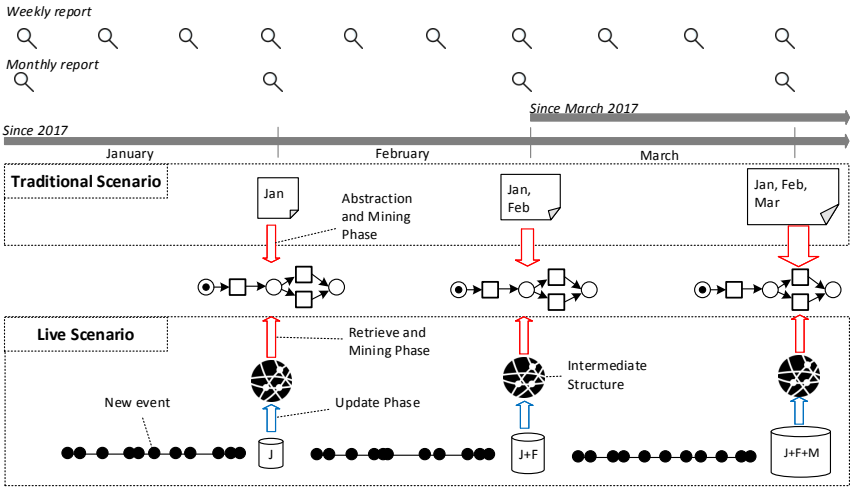

Fig. 1. Traditional vs live scenario in process discovery

of reporting requires an analyst to repeatedly produce analysis results from data that grows over time.

Existing process mining tools are not tailored towards such recurrent analyses. Instead, they require the analysts to export the event data from a running system, import it to the mining tool which pre-processes the data during importing and then use the tool on the pre-processed data. As the amount of data grows, the import and pre-processing phase takes longer and longer causing the analyst to waste time.

However, most information systems nowadays record real-time event data about business processes during their executions. This enables analysis techniques to import and pre-process the data when it "arrives". By shifting the pre-processing time an analyst is able to do process mining on the pre-processed data instantly.

The idea of this live scenario is sketched in Figure 1. In the live scenario, we introduce a persistent storage for various structures that are kept as "intermediate structures" by process mining algorithms. We then show how such intermediate structures can be updated without the need for full recomputation every time an event arrives. Using intermediate structures of a wide range of process mining techniques, we show the feasibility of our approach in the general process mining setting and using experiments on real-life data, we show the added time-benefit for the analyst.

This paper is organized as follows. Section 2 discusses some related work. Then, in Section 3 we introduce recurrent process mining and we focus on two traditional techniques in this setting. In Section 4, we show how recurrent process mining can be performed on live event data and we prove that the two techniques of Section 3 can be made incremental. We show the improvements of our work using real-life data in Section 5. Finally, we conclude the paper in Section 6.

2 Related Work

For a detailed explanation about process mining, we refer to [17]. Here we primarily focus on the work related to recurrent process discovery in process mining and its applications on live event data.

In the current setting of process discovery, event data from a file-based system is imported to a process mining tool. This technique potentially creates redundancy of data reloading in environments which necessitate some repetitions in the discovery. Therefore, some researches have been looking to the area of databases, Hadoop, and other ways to store event data in a persistent manner.

A study in [23] examined database called XESame. To access the data in XESame, one needs to materialize the data by selecting and matching it with XES [4] elements. It does not provide a direct access to the database. A more advanced technique using ontology was proposed in [1]. In this work, data can be accessed on-demand using query unfolding and rewriting techniques, called ontology-based data access. However, the performance issues make this approach unsuitable for large event logs.

Relational XES, or RXES, was introduced in [19]. RXES schema was designed with a focus on XES standard. Using experiments with real life data, it was shown that RXES typically uses less memory compared to the file-based OpenXES and MapDB XESLite implementations [11]. As an improved version of RXES, DB-XES was introduced in [16]. Besides the basic schema, it was extended with directly-follows relation which enables some process discovery techniques, such as the Inductive Miner [6, 7] and the Alpha Miner [17].

Process mining not only covers procedural process discovery, but also declarative process discovery. The work in [14] deals with declarative process discovery using SQL databases. Building on top of RXES, the authors introduce a mining approach that directly works on relational event data by querying the log with conventional SQL. Queries can be customised and cover process perspective beyond the control flow, e.g., organisational aspects. However, none of these techniques handles live event data, the focus is often on static data that has been imported in a database.

Treating event data as a dynamic sequence of events has been explored in [20, 21]. This work presented single-node stream extension of process mining tool ProM which enables researchers and business users to also apply process mining on streaming-data. The applicability of the framework on the cooperative aspects of process mining was demonstrated in [22]. Here the authors define the notion of case administration to store for each case the (partial) sequence of (activity, resource)-pairs seen so far.

Another online process discovery based on streaming technology was proposed in [8]. This work presented a novel framework for the discovery of LTL-based declarative process models from streaming event data. The framework continuously updates a set of valid business constraints based on the events occurring in the event stream. Moreover, it gives the user meaningful information about the most significant concept drifts occurring during process execution.

However, from the real-time techniques using streaming event data that we have seen so far, none of them deals with recurrent process discovery. Intermediate results are not stored after presenting the results. Therefore, all data (old and new) needs to be reloaded and processed each time a new analysis is needed. Building on from that

concern, this work explores both in the ability to process live event data and to handle recurrent questions in process discovery.

3 Recurrent Process Mining

The challenge in process mining is to gain insights into an operational system in which activities are executed. The execution of these activities is reflected through events, i.e. events are occurrences of activities. Furthermore, in process mining, activities are being executed in the context of cases (also referred to as process instances) at a particular point in time.

Definition 1 (Events, cases, and activities).
Let Σ be the universe of activities, let C be the universe of cases, and let Υ be the universe of events. We define $\#_{act} : \Upsilon \rightarrow \Sigma$ a function labeling each event with an activity. We define $\#_{cs} : \Upsilon \rightarrow C$ a function labeling each event with a case. We define $\#_{tm} : \Upsilon \rightarrow \mathbb{R}$ a function labeling each event with a timestamp.

In process mining, the general assumption is that event data is provided in the form of an event log. Such an event log is basically a collection of events occurring in a specific time period, in which the events are grouped by cases sequentialized based on their time of occurrence.

Definition 2 (Event log and trace).
Let $E \subseteq \Upsilon$ be a collection of events and $t_s, t_e \in \mathbb{R}$ two timestamps with $t_s < t_e$ relating to the start and the end of the collection period.

A trace $\sigma \in E^*$ is a sequence of events such that the same event occurs only once in σ, i.e. $|\sigma| = |\{e \in \sigma\}|$. Furthermore, each event refers in a trace refers to the same case $c \in C$, i.e. $\forall_{e \in \sigma} \#_{cs}(e) = c$ and we assume all events within the given time period are included, i.e. $\forall_{e \in \Upsilon}(\#_{cs}(e) = c \wedge t_s \leqslant \#_{tm}(e) \leqslant t_e) \implies e \in \sigma$.

An event log $L \in \wp(E^*)$[1] is a set of traces.

Note that the time-period over which an event log is collected plays an important role in this paper. In most process mining literature, this time period is neglected and the event log is assumed to span eternity. However, in practice, analysis always consider event data pertaining to a limited period of time. Therefore, in this paper, we explicitly consider the following process mining scenarios (as depicted in Figure 1):

- Analysts perform process mining tasks on a recurrent schedule at regular points, e.g. once a week about the last week, or once a month about the last month, or
- Analysts perform process mining on the data since a pre-determined point in time, e.g. since 2017, or since March 2017.

[1] $\wp(E^*)$ denotes a powerset of sequences, i.e. $L \subseteq E^*$

3.1 Traditional Recurrent Process Mining

To execute the two process mining scenarios, a multitude of process mining techniques is available. However, all have two things in common, namely (1) that the starting point for analysis is an event log and (2) that the analysis is performed in three phases, namely: *loading*, *abstraction*, and *mining*.

While the details may differ, all process mining techniques build an *intermediate structure* in memory during the abstraction phase. Typically, the time needed to execute this phase is linear in the number of events in the event log. This intermediate structure (of which the size is generally polynomial in the number of activities in the log, but not related to the number of events anymore) is then used to perform the actual mining task. In this paper, we consider two traditional process mining techniques in more detail, with different characteristics.

First, the Inductive Miner [17] is considered, which is known to be flexible, to have formal guarantees, and to be scalable. For the Inductive Miner, the intermediate structure is the so-called *direct succession relation* (Definition 3) which counts the frequencies of direct successions.

Definition 3 (Direct succession [17]).
Let L be an event log over $E \subseteq \Upsilon$. The direct succession relation $\twoheadrightarrow_L : \Sigma \times \Sigma \to \mathbb{N}$ counts the number of times activity a is directly followed by activity b in some cases in L as follows:

$$\twoheadrightarrow_L(a,b) = \Sigma_{\sigma \in L} \Sigma_{i=1}^{|\sigma|-1} \begin{cases} 1, & \text{if } \#_{act}(\sigma(i)) = a \ \wedge \ \#_{act}(\sigma(i+1)) = b \\ 0, & \text{otherwise.} \end{cases}$$

Second, a technique called MINERful [3] is considered. This technique focuses on discovering declarative models [9, 10], using a language that was first introduced in [12]. MINERful has demonstrated the best scalability with respect to the input size compared to the other declarative discovery techniques [3]. The abstraction phase in MINERful computes a number of intermediate structures on the event log which are then used during mining.

The intermediate structures used by MINERful are defined as follows:

Definition 4 (MINERful relations [2]).
Let L be an event log over $E \subseteq \Upsilon$. The following relations are defined for MINERful:

$\#_L : \Sigma \to \mathbb{N}$ *counts the occurrences of activity a in event log L, i.e.*
$\#_L(a) = |\{e \in E \mid \#_{act}(e) = a\}|$.

$\not\twoheadrightarrow_L : \Sigma \times \Sigma \to \mathbb{N}$ *counts the occurrences of activity a with no following b in the traces of L, i.e.*

$$\not\twoheadrightarrow_L(a,b) = \Sigma_{\sigma \in L} \Sigma_{i=1}^{|\sigma|} \begin{cases} 1, & \text{if } a \neq b \ \wedge \ \#_{act}(\sigma(i)) = a \ \wedge \\ & \quad \forall j,\ i < j \leqslant |\sigma|,\ \#_{act}(\sigma(j)) \neq b \\ 0, & \text{otherwise} \end{cases}$$

$\not\rightarrow_L : \Sigma \times \Sigma \rightarrow \mathbb{N}$ *counts the occurrences of* a *with no preceding* b *in the traces of* L*, i.e.*

$$\not\rightarrow_L(a,b) = \Sigma_{\sigma \in L} \Sigma_{i=1}^{|\sigma|} \begin{cases} 1, & \text{if } a \neq b \ \wedge \ \#_{act}(\sigma(i)) = a \ \wedge \\ & \forall j,\ 1 \leqslant j < i,\ \#_{act}(\sigma(j)) \neq b \\ 0, & \text{otherwise} \end{cases}$$

$\not\nrightarrow_L : \Sigma \times \Sigma \rightarrow \mathbb{N}$ *counts the occurrences of* a *with no co-occurring* b *in the traces of* L*, i.e.*

$$\not\nrightarrow_L(a,b) = \Sigma_{\sigma \in L} \Sigma_{i=1}^{|\sigma|} \begin{cases} 1, & \text{if } a \neq b \ \wedge \ \#_{act}(\sigma(i)) = a \ \wedge \\ & \forall j,\ 1 \leqslant j \leqslant |\sigma|,\ \#_{act}(\sigma(j)) \neq b \\ 0, & \text{otherwise} \end{cases}$$

$\not\rightsquigarrow_L : \Sigma \times \Sigma \rightarrow \mathbb{N}$ *counts how many times after an occurrence of* a*,* a *repeats until the first occurrence of* b *in the same trace. if no* b *occurs after* a*, then the repetitions after* a *are not counted , i.e.*

$$\not\rightsquigarrow_L(a,b) = \Sigma_{\sigma \in L} \Sigma_{i=1}^{|\sigma|} \begin{cases} 1, & \text{if } a \neq b \ \wedge \ \#_{act}(\sigma(i)) = a \ \wedge \\ & \exists j,\ i < j \leqslant |\sigma|,\ \#_{act}(\sigma(j)) = b \ \wedge \\ & \exists k,\ 1 \leqslant k < i,\ \#_{act}(\sigma(k)) = a \ \wedge \\ & \forall l,\ k < l < i,\ \#_{act}(\sigma(l)) \neq b \\ 0, & \text{otherwise} \end{cases}$$

$\not\leftsquigarrow_L : \Sigma \times \Sigma \rightarrow \mathbb{N}$ *is similar to* $\not\rightsquigarrow$ *but reading the trace backwards, i.e.* $\not\leftsquigarrow_L(a,b) = \not\rightsquigarrow_{L'}(a,b)$ *where* L' *is such that all traces in* L *are reversed.*

To illustrate how both mining algorithms use the relations described here, we refer to Table 1. Here, we show (by example), how both mining techniques use the relations as intermediate structures to come to a result. In order to apply these traditional techniques in a recurrent setting, the loading of the data, the abstraction phase, and the mining phase have to be repeated. When over time the event data grows, the time to execute the three phases also grows, hence performing the recurrent mining task considering one year of data will take 52 times longer than considering one week of data.

In Table 2 we show an example of the two techniques applied to a real-life dataset of the BPI Challenge 2017 [18] which contains data of a full year. We record the times to perform the three phases of importing, abstraction, and mining on this dataset after the first month, at the middle of the year, and at the end of the year. It is clear that indeed the importing and abstraction times grow considerably, while the actual mining phase is orders of magnitude faster.

Figure 2 shows the result of the Inductive Miner after the first 5 weeks of data, i.e. if an analyst has produced this picture on January 29^{th} 2016, it would have taken 3.7305 seconds ($= 0.8520 + 2.8531 + 0.0254$) in total to load the event log, build the

IM:	A sequence cut of L is a cut $(\rightarrow, A_1, ..., A_n)$ such that $\forall i, j \in \{1, ..., n\}\ \forall a \in A_i\ \forall b \in A_j\ i < j \Rightarrow (\not\rightsquigarrow^+(L,a,b) \ \wedge \ \neg \not\rightsquigarrow^+(L,b,a))$
MF:	$\mathit{ChainResponse}(L,a,b) = \frac{\not\rightsquigarrow(L,a,b)}{\#(L,a)}$

Table 1: Examples of the use of intermediate structures in Inductive Miner (IM) and MINERful (MF)

Week	MINERful			Inductive Miner		
	Load	Abstr	Mining	Load	Abstr	Mining
5	0.7789	2.3350	21.2563	0.8520	2.8531	0.0254
26	4.2535	24.9009	29.1813	3.6319	30.9528	0.0257
52	9.0895	60.5857	32.0457	9.5854	93.5118	0.0291

Table 2: Process mining times (in seconds) in the traditional setting on data of the BPI Challenge 2017 [18]

abstraction, and do the mining in order to produce this picture using the Inductive Miner in ProM.

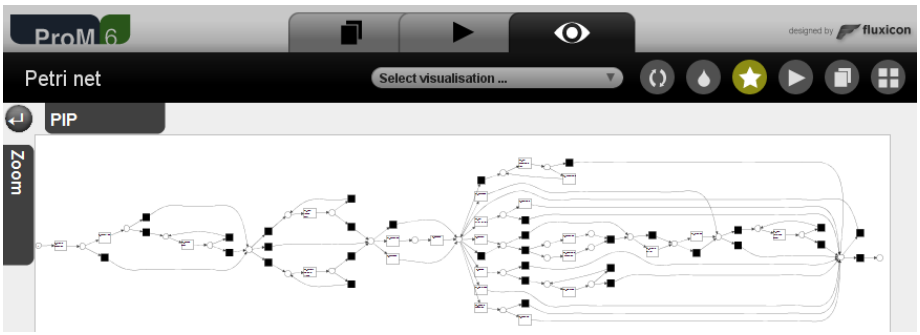

Fig. 2. Screenshot of ProM showing the result of the Inductive Miner on the BPI Challenge 2017 data considering the first 5 weeks of data.

In Section 4, we present a method to store the intermediate structures in a persistent storage and to keep them live under updates, i.e. every time an event is generated, the intermediate structure is updated (in constant time). This way, when a process mining task is performed, the time spent by the analyst is limited to the time to retrieve the intermediate structure and to do the actual mining.

4 Recurrent Process Mining with Live Event Data

It is easy to see that given an event log L, the relation in Definition 3 can be computed during a single linear pass over the event log, visiting each event exactly once. In this section, we present two live process mining techniques, based on the Inductive Miner and MINERful, which do not require the analyst to repeat the import and analysis phases every time a process mining task is performed.

In order to enable live process mining, we use a persistent event storage, called DB-XES [16], which uses a relational database to store event data. The full structure of this relational database is beyond the scope of this paper, but what is important is that given a trace, it is possible to quickly retrieve the last event recorded for that trace.

Definition 5 (Last event in a trace).
Let $E \subseteq \Upsilon$ be a collection of events and let $c \in C$ be a case. The function $\lambda : C \rightarrow E \cup \{\perp\}$ is a function that returns the last event in E belonging to case c, i.e.

$$\lambda(c) = \begin{cases} \perp, & \text{if } \forall_{e \in E} \#_{cs}(e) \neq c, \\ e \in E, & \text{if } \#_{cs}(e) = c \wedge \nexists e' \in E(\#_{cs}(e') = c \wedge \#_{tm}(e') > \#_{tm}(e)). \end{cases}$$

Using DB-XES as a persistent storage and making use of the ability to query for the last event in a trace, we present the Incremental Inductive Miner in Section 4.1 and Incremental MINERful in Section 4.2

4.1 Incremental Inductive Miner

The Inductive Miner uses only the frequency of direct successions between activities as input as defined in Definition 3. Therefore, to enable an incremental version of the Inductive Miner, we present an update strategy that, given the relation $\twoheadrightarrow_L$ for some log L and a new event e, we can derive the relation $\twoheadrightarrow_{L'}$ where L' is the log L with the additional event e.

Theorem 1 (Updating relation $\twoheadrightarrow$ is possible).
Let $E \subseteq \Upsilon$ be a set of events and L a log over E. Let $e \in \Upsilon \backslash E$ be a fresh event to be added such that for all $e' \in E$ holds $\#_{ts}(e') < \#_{ts}(e)$ and let $E' = E \cup \{e\}$ be the new set of events with L' the corresponding log over E'. We know that for all $a, b \in \Sigma$ holds that:

$$\twoheadrightarrow_{L'}(a,b) = \twoheadrightarrow_L(a,b) + \begin{cases} 0 & \text{if } \lambda(\#_{cs}(e)) = \perp, \\ 1 & \text{if } \#_{act}(\lambda(\#_{cs}(e))) = a \wedge \#_{act}(e) = b, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Let $c = \#_{cs}(e) \in C$ be the case to which the fresh event belongs.

If for all $e' \in E$ holds that $\#_{cs}(e') \neq c$, then this is the first event in case c and we know that $L' = L \cup \langle e \rangle$. Hence relation $\twoheadrightarrow$ does not change from L to L' as indicated by case 1.

If there exists $e' = \lambda(\#_{cs}(e)) \in E$ with $\#_{cs}(e') = c$, then we know that there is a trace $\sigma_c \in L$. Furthermore, we know that $L' = (L \backslash \{\sigma_c\}) \cup \{\sigma_c \cdot \langle e \rangle\}$, i.e. event e gets added to trace σ_c of L. This implies that $\twoheadrightarrow_{L'}(\#_{act}(e'), \#_{act}(e)) = \twoheadrightarrow_L(\#_{act}(e'), \#_{act}(e)) + 1$ as indicated by case 2.

In all other cases, the number of direct successions of two activities is not affected. ■

Using the simple update procedure indicated in Theorem 1 the Incremental Inductive Miner allows for recurrent process mining under live updates. In Section 5 we show how the effect of keeping relation $\twoheadrightarrow$ live on the total time needed to perform the recurrent process mining task. However, we first introduce Incremental MINERful for which the other relations of Definition 4 need to be kept live.

4.2 Incremental MINERful

Keeping the direct succession relation live under updates is rather trivial. Most relations of Definition 4 however do not allow for a simple update strategy. Therefore, we introduce a so-called *controller function* which we keep live under updates. Then we show that, using the controller function, we can keep all relations live under updates.

Definition 6 (Controller function).
Let $E \subseteq \Upsilon$ be a set of events and L a log over E. Let $\sigma_c \in L$ be a trace in the log referring to case $c \in C$. $\not\vdash^c_L : \Sigma \times \Sigma \to \mathbb{N}$ is a controller function such that for all $a, b \in \Sigma$ holds that:

$$\not\vdash^c_L(a,b) = \Sigma_{i=1}^{|\sigma_c|} \begin{cases} 1, & \text{if } \#_{act}(\sigma_c(i)) = a \wedge \\ & (a = b \ \vee \forall j,\ i < j \leqslant |\sigma_c|,\ \#_{act}(\sigma_c(j)) \neq b) \\ 0, & \text{otherwise.} \end{cases}$$

$\not\vdash^c_L(a,b)$ counts the occurrences of $a \in \Sigma$ with no following $b \in \Sigma$ in σ_c if $a \neq b$. If $a = b$ then it counts the occurrence of a in σ_c.

The controller function $\not\vdash^c$ of Definition 6 is comparable to relation $\not\vdash$ of Definition 4. However, $\not\vdash^c$ is defined on the case level, rather than on the log level, i.e. in our persistent storage, we keep the relation $\not\vdash^c$ for each case in the set of events. In many practical situations, it is known when a case is finished, i.e. when this relation can be removed from the storage.

Using the controller function, we show how all relations of Definition 4 can be kept live under updates. To prove this, we first show that we can keep the controller function itself live under updates and then we show that this is sufficient.

Lemma 1 (Updating controller function is possible).
Let $E \subseteq \Upsilon$ be a set of events and L a log over E. Let $e \in \Upsilon \backslash E$ be a fresh event to be added such that for all $e' \in E$ holds $\#_{ts}(e') < \#_{ts}(e)$ and let $E' = E \cup \{e\}$ be the new set of events with L' the corresponding log over E'. Furthermore, let $c = \#_{cs}(e) \in C$ be the case to which the fresh event belongs. We know that for all $a, b \in \Sigma$ holds that:

$$\not\vdash^c_{L'}(a,b) = \begin{cases} \not\vdash^c_L(a,b) + 1 & \text{if } a = \#_{act}(e), \\ 0 & \text{if } a \neq \#_{act}(e) \wedge b = \#_{act}(e), \\ \not\vdash^c_L(a,b) & \text{otherwise.} \end{cases}$$

Proof. Let $\sigma_c \in L$ be the trace corresponding to case c in L, let $\sigma'_c = \sigma_c \cdot \langle e \rangle \in L'$ be the trace corresponding to case c in L' and let $x = \#_{act}(e) \in \Sigma$ be the activity label of e.

Clearly, for all $e' \in \sigma_c$ holds that e is a succeeding event with label x, hence $\not\vdash^c_{L'}(a,x) = 0$ for all $a \neq x$ (case 2). Also, since e is the last event in the trace, the number of times activity x is not followed by any other label $a \in \Sigma$, $a \neq x$, in σ'_c is one more than before (case 1). Furthermore, the occurrence count of x is also increased by one (case 1). Finally, the relations between all pairs not involving activity x is not changed (case 3). ■

Figure 3 provides an example where $\not\vdash^c$ is kept updated under insertion of each event in a trace. The trace considered is $\sigma_c = \langle a, a, b \rangle$. In each step, the values in the

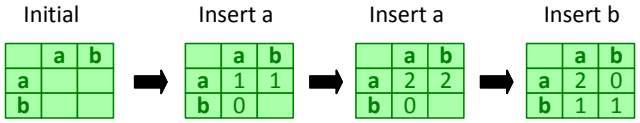

Fig. 3. An example of updating controller function $\triangleright^c$. Events (which are represented by their activity names) in trace $\sigma_c = \langle a, a, b \rangle$ are inserted one by one and in each insertion values under $\triangleright^c$ are updated, assuming $\Sigma = \{a, b\}$.

row and column corresponding to the activity label that is being inserted are updated. The rationale behind *adding one to the row* (case 1) is that a new insertion of an activity x in a trace σ_c increases the occurrences of x in σ_c with no other activities succeeding it, since x is the current last activity of σ_c. While *reseting the column* (case 2) means that the insertion of x invalidates the occurrences of activities other than x with no following x.

The complexity of the update algorithm is linear in the number of activities as for each event all other activities need to be considered in the corresponding row and column. This makes the update procedure slightly more complex than the updating of the direct succession for the Incremental Inductive Miner as the latter only has to consider the last label in the trace of the new event.

Lemma 2 (Updating MINERful relations is possible).
Let $E \subseteq \Upsilon$ be a set of events and L a log over E. Let $e \in \Upsilon \setminus E$ be a fresh event to be added such that for all $e' \in E$ holds $\#_{ts}(e') < \#_{ts}(e)$ and let $E' = E \cup \{e\}$ be the new set of events with L' the corresponding log over E'. Furthermore, let $c = \#_{cs}(e) \in C$ be the case to which the fresh event belongs.

Updating $\triangleright^c_L$ to $\triangleright^c_{L'}$ is sufficient to update the relations $\#, \triangleright, \triangleleft, \nparallel, \not\rightarrow$, and $\not\leftarrow$ in the following way for all $a, b \in \Sigma$:

$$\#_{L'}(a) = \begin{cases} \#_L(a) + 1 & \text{if } a = \#_{act}(e), \\ \#_L(a) & \text{otherwise} \end{cases}$$

$$\triangleright_{L'}(a,b) = \triangleright_L(a,b) + \begin{cases} -\triangleright^c_L(a,b) + \triangleright^c_{L'}(a,b) & \text{if } a \neq b, \\ 0 & \text{otherwise} \end{cases}$$

$$\triangleleft_{L'}(a,b) = \triangleleft_L(a,b) + \begin{cases} 1 & \text{if } a \neq b \ \wedge \ a = \#_{act}(e) \wedge \triangleright^c_{L'}(b,b) = 0, \\ 0 & \text{otherwise} \end{cases}$$

$$\nparallel_{L'}(a,b) = \nparallel_L(a,b) + \begin{cases} 1 & \text{if } a \neq b \ \wedge a = \#_{act}(e) \wedge \triangleright^c_{L'}(b,b) = 0, \\ -\triangleright^c_L(a,a) & \text{if } a \neq b \ \wedge b = \#_{act}(e) \wedge \triangleright^c_{L'}(b,b) = 1, \\ 0 & \text{otherwise} \end{cases}$$

$$\not\rightarrow_{L'}(a,b) = \not\rightarrow_L(a,b) + \begin{cases} \triangleright^c_L(a,b) \ - \ 1 & \text{if } a \neq b \ \wedge \ b = \#_{act}(e) \wedge \\ & \triangleright^c_L(a,b) \geqslant 1, \\ 0 & \text{otherwise} \end{cases}$$

$$\nleftarrow\!\!\rho_{L'}(a,b) \;=\; \nleftarrow\!\!\rho_{L}(a,b) + \begin{cases} 1 & \text{if } a \neq b \;\wedge\; a = \#_{act}(e) \;\wedge \\ & \not\vdash^{c}_{L}(b,b) \geqslant 1 \wedge \not\vdash^{c}_{L}(a,b) \geqslant 1, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. The full technical proof is omitted because of space limitation. However, the intuition behind the proof is as follows:

$\#(a)$ The sum of occurrences in the log can be updated trivially when adding an event.

$\not\vdash(a,b)$ The occurrence of a with no following b in the log is only affected by the case c to which e belongs, hence the update here is the same as for the controller function if $a \neq b$.

$\not\dashv(a,b)$ The occurrence of a with no preceding b is only affected if a is inserted in a trace in which b did not occur yet.

$\not\dashv\!\not\vdash(a,b)$ The occurrence of a with no co-occurring b is only affected if a is inserted in a trace in which b did not occur yet. Then, the value is reduced by the occurrence of a when b is inserted in the trace at the first time.

$\rightsquigarrow(a,b)$ The repetition of a until b is only affected if b is added and, before adding, a was occurring at least once since the previous b, i.e. since the last time the column of b was reset.

$\nleftarrow\!\!\rho(a,b)$ The repetition of a until b when reading the trace contrariwise is only affected if a is inserted in the trace, b appeared earlier in the trace, and the number of times a was repeated since then is at least once.

■

Finally, using the controller function, we can show that the MINERful relations can be kept live under updates.

Theorem 2 (Updating all MINERful relations is possible).
Relations $\#, \not\vdash, \not\dashv, \not\dashv\!\not\vdash, \twoheadrightarrow, \rightsquigarrow$, *and* $\nleftarrow\!\!\rho$ *can be kept live under insertion of new events.*

Proof. Relation $\twoheadrightarrow$ can be kept live under update (Theorem 1). It is possible to incrementally update the controller function $\not\vdash^{c}$ for each insertion of a new event (Lemma 1). Updating $\not\vdash^{c}$ is sufficient to update the intermediate structures $\#, \not\vdash, \not\dashv, \not\dashv\!\not\vdash, \rightsquigarrow$, and $\nleftarrow\!\!\rho$ (Lemma 2). Therefore, it is possible to keep those intermediate structures up-to-date in each insertion of a new event. ■

In Table 3 we revisit the scenario's of Table 2 and we show the time needed to perform the retrieval of the relations from the persistent storage, as well as the mining time for both incremental algorithms. It is clear that the retrieval time considerably smaller than the time to import all events and perform the abstraction, while the mining time is comparable in both cases (the differences for MINERful are explained by the fact that we use two different implementations for the mining phase, while for the Inductive Miner, the mining phase is performed using the same implementation).

As Table 3 shows, we can achieve great speed increases when keeping the intermediate structures in a persistent storage. It is important to realize however that these intermediate structures need to be kept *for each time period of interest*, i.e. if we refer back to Figure 1, then we see that these structures are to be kept for the current week, the current month, since 2017 and since March 2017.

Week	MINERful	Inductive Miner	Incremental MF		Incremental IM	
	Total	Total	Retrieve	Mining	Retrieve	Mining
5	24.3702	3.7305	0.0809	0.0160	0.0116	0.0238
26	58.3357	34.6104	0.0835	0.0172	0.0127	0.0238
52	101.7209	103.1263	0.0883	0.0184	0.0132	0.0260

Table 3: Process mining times (in seconds) of the traditional setting compared to the live setting on data of the BPI Challenge 2017 [18]

5 Experimental Results

We implemented the two algorithms presented as ProM[2] plug-ins called *Database-Incremental Inductive Miner (DIIM)*[3] and *Database-Incremental Declare Miner (DIDM)*[4]. Both DIIM and DIDM are designed for recurrent process discovery based on live event data. DIIM and DIDM use DB-XES as the back-end storage and the Inductive Miner or MINERful as the mining algorithm respectively. The current implementation of DIDM is able to discover the following constraints: *RespondedExistence, Response, AlternateResponse, ChainResponse, Precedence, AlternatePrecedence, ChainPrecedence, CoExistence, Succession, AlternateSuccession, ChainSuccession, NotChainSuccession, NotSuccession,* and *NotCoExistence*. In this section, we show the experimental results of applying DIIM and DIDM in a live scenario and we compare it to traditional Inductive Miner and MINERful.

For the experiment, we used a real dataset from BPI Challenge 2017 [18]. This dataset pertains to the loan applications of a company from January 2016 until February 2017. In total, there are 1,202,267 events and 26 different activities which pertain to 31,509 loan applications.

In this experiment, we looked into some weekly reports where we were interested to see both procedural and declarative process models of the collective data since 2016. The last working day, i.e. Friday, was chosen as the day when we performed the process discovery to have a progress report for that week. In live scenario, we assumed that each event was inserted to the DB-XES database precisely at the time stated in the timestamp attribute of the event log. Then, the DB-XES system immediately processed each new event data as it arrived using triggers in the relational database, implementing the update procedures detailed in Section 4, thus keeping the relations as well as the controller function live under updates. In traditional scenario, we split the dataset into several logs such that each log contained data for one week. For the n-th report, we combined the log from the first week until the n-th week, loaded it into ProM, and discovered a process model.

Figure 4 shows the experimental results. The x-axis represents the n-th week, while the y-axis represents the time spent by user (in seconds) to discover procedural and

[2] See `http://www.processmining.org` and `http://www.promtools.org`

[3] `https://svn.win.tue.nl/repos/prom/Packages/DatabaseInductiveMiner/Trunk/`

[4] `https://svn.win.tue.nl/repos/prom/Packages/MixedParadigm/Trunk/`

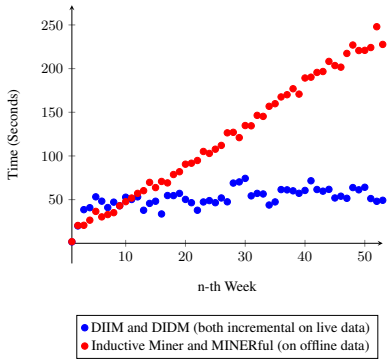

Fig. 4. The comparison of recurrent process discovery using DIIM and DIDM vs traditional Inductive Miner and MINERful

declarative process models. The blue dots are the experiment using DIIM and DIDM which includes the total times to insert new events, update the intermediate structures, retrieve the values from the DB-XES, and mine the process models, while the red dots are the experiment using traditional Inductive Miner and MINERful which includes the time to load the XES event logs, build the intermediate structures, and mine the process models.

As shown in the Figure 4, after the first two months, our incremental techniques became faster, even when considering the time needed to insert events in the relational database, a process that is typically executed in real time and without the business analyst being present. More important however, is the trendlines of both approaches.

As expected, the time to perform the process mining task in the traditional setting is growing linear in the size of the event data (the arrival rate of events in this dataset is approximately constant during the entire year). This is due to the fact that the first two phases of loading the data and doing the abstraction into the intermediate structures scales linearly in the number of events, whereas the mining scales in the number of activities. The latter is considerably smaller than the former in most practical cases as well as in this example. Our incremental algorithms are more stable over time as the update time only depends on the number of *newly inserted* events and both the retrieval and mining times depend on the number of activities rather than the number of events.

The variations in the recorded values of the DIIM and DIDM are therefore explained by the number of inserted events in a day. The higher the number of newly inserted events, the longer it takes to do the update in the relational database system of the intermediate structures. However, the total update time remains limited to around 1 minute per day.

In order to see the average time for doing an update for a single event, we normalized the total update time with the number of events inserted in each day as shown in Figure 5. The x-axis represents the n-th day, while the y-axis represents the update time per event. As shown from the Figure 5, the update time in the first week was lower than the update time in later weeks. This effect is explained by the fact that the update procedure

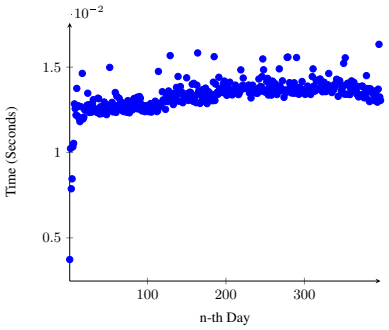

Fig. 5. Average update time per event

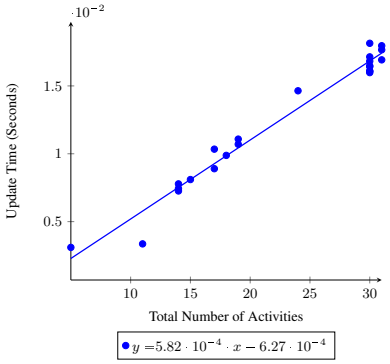

Fig. 6. The influence of number of activities to update time

for the controller function is linear in the number of activities in the log (as discussed under Lemma 1). During the first week, not all activities have been recorded yet and hence the update times are lower. However, after about one week, all activities have been seen and the average time to conduct an update for a single event stabilizes around 0.013 seconds, i.e. the database system can handle around 75 events per second and this includes the insertion of the actual event data in the underlying DB-XES tables.

To validate the fact that the update time scales linearly in the number of activities, we refer to Figure 6. For this experiment, we used a different dataset with 31 different activities and eleven thousands of events, provided to us by Xerox Services, India. The x-axis represents the total number of different activities which has been inserted to the database, while the y-axis represents the time in seconds to update an event. From the figure, it is clear that the update time indeed depends linearly on the number of activities.

It is important to realize that the results of the process mining techniques in both the traditional and the live setting are not different, i.e. the process models are identical. Figure 7 shows a screenshot of a process model in ProM, produced by the Incremental Inductive Miner considering all the data in the original file. In the traditional setting, it would have taken an analyst 103.1263 seconds to load the event log, build the abstraction and do the mining in order to get this picture on December 30^{th} 2016. Due to the availability of the intermediate structure in the database, it would take the analyst only 0.0392 seconds to produce the same result using the Incremental Inductive Miner.

6 Conclusion

Process mining aims to give insights into processes by discovering process models which adequately reflect the behavior seen in an event log. One of the challenges in process mining is the recurrent nature of mining tasks which, using traditional tools and techniques, require analysts to import and pre-process larger and larger datasets.

In this paper we focused on recurrent process discovery on live event data. Using two process mining techniques as examples, we show how we can reduce the time

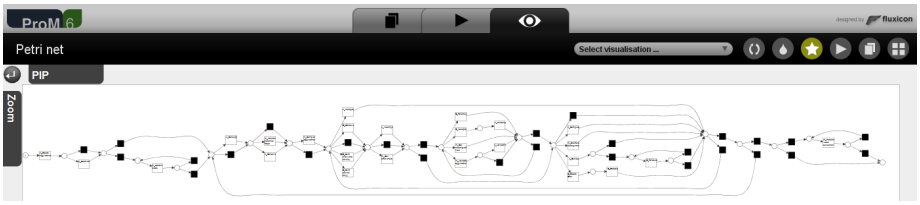

Fig. 7. Screenshot of ProM showing the result of the Inductive Miner on the BPI Challenge 2017 data considering all data.

needed for an analyst to do process mining by storing intermediate structures in a persistent storage.

Using two algorithms, for procedural control flow discovery and declarative control flow discovery, we show how these can be adopted to work with live intermediate structures. In the former case this intermediate structure is nothing more than a direct succession relation with frequencies, which is trivial to keep up-to-date. In the latter case however, we require some additional information to be kept in the persistent storage for each currently open case in order to quickly produce the required relations.

Using a concrete implementation, we use the relational database called DB-XES to store the event data and the intermediate structures. In this relational database, we added triggers to update all intermediate structures with the insertion of each event and we implemented the Incremental Inductive Miner and the Incremental MINERful as versions of existing techniques which are able to use these persistent intermediate structures directly.

We tested the performance of the proposed approach and compared it to the traditional techniques using real-life datasets. We show that loading and mining time of the traditional approaches grow linearly as the event data grows. In contrast, our incremental implementations show constant times for updating (per event) and the retrieval and mining times are independent of the size of the underlying data.

The core ideas in the paper are not limited to control flow. They are, for example, trivially extended to store intermediate structures keeping track of average times between activities or for social networks. A more fundamental challenge for future work is the updating of the intermediate structures in batches of events, rather than for each event separately. Furthermore, we aim to enable these techniques to keep these structures live under removal of past events.

References

1. D. Calvanese, M. Montali, A. Syamsiyah, and W.M.P. van der Aalst. Ontology-Driven Extraction of Event Logs from Relational Databases. In *BPI 2015*, pages 140–153, 2015.
2. C. Di Ciccio, F.M. Maggi, and J. Mendling. Efficient Discovery of Target-Branched Declare Constraints. *Information Systems*, 56:258 – 283, 2016.
3. C. Di Ciccio and M. Mecella. *Mining Constraints for Artful Processes*, pages 11–23. Springer Berlin Heidelberg, 2012.
4. C.W. Günther. XES Standard Definition. www.xes-standard.org, 2014.

5. M.J. Jans, M. Alles, and M.A. Vasarhelyi. Process Mining of Event Logs in Auditing: Opportunities and Challenges. *Available at SSRN 2488737*, 2010.
6. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In *Petri Nets 2013*, pages 311–329, 2013.
7. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *BPM Workshop 2013*, pages 66–78, 2013.
8. F.M. Maggi, A. Burattin, M. Cimitile, and A. Sperduti. *Online Process Discovery to Detect Concept Drifts in LTL-Based Declarative Process Models*, pages 94–111. Springer Berlin Heidelberg, 2013.
9. F.M. Maggi, M. Dumas, L. García-Bañuelos, and M. Montali. Discovering Data-Aware Declarative Process Models from Event Logs. In *BPM 2013*, pages 81–96, 2013.
10. F.M. Maggi, A.J. Mooij, and W.M.P. van der Aalst. User-Guided Discovery of Declarative Process Models. In *CIDM 2011*, pages 192–199, 2011.
11. F. Mannhardt. XESLite Managing Large XES Event Logs in ProM. *BPM Center Report BPM-16-04*, 2016.
12. M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users*. PhD thesis, TU Eindhoven, 2008.
13. E. Rojas, J. Munoz-Gama, M. Sepúlveda, and D. Capurro. Process Mining in Healthcare: A Literature Review. *Journal of Biomedical Informatics*, 61:224–236, 2016.
14. S. Schönig, A. Rogge-Solti, C. Cabanillas, S. Jablonski, and J. Mendling. *Efficient and Customisable Declarative Process Mining with SQL*, pages 290–305. Springer International Publishing, Cham, 2016.
15. S. Suriadi, M.T. Wynn, C. Ouyang, A.H.M. ter Hofstede, and N.J. van Dijk. Understanding Process Behaviours in a Large Insurance Company in Australia: A Case Study. In *CAiSE 2013*, pages 449–464, 2013.
16. A. Syamsiyah, B.F. van Dongen, and W.M.P. van der Aalst. DB-XES: Enabling Process Mining in the Large. In *SIMPDA 2016*, pages 63–77, 2016.
17. W.M.P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.
18. B.F. van Dongen. BPI Challenge 2017, 2017.
19. B.F. van Dongen and S. Shabani. Relational XES: Data Management for Process Mining. In *CAiSE 2015*, pages 169–176, 2015.
20. S.J. van Zelst, A. Burattin, B.F. van Dongen, and H.M.W. Verbeek. Data Streams in ProM 6: A Single-Node Architecture. In *BPM Demo Session 2014*, page 81, 2014.
21. S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. Know What You Stream: Generating Event Streams from CPN Models in ProM 6. In *BPM Demo Session 2015*, pages 85–89, 2015.
22. S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. Online Discovery of Cooperative Structures in Business Processes. In *OTM Conferences 2016*, pages 210–228, 2016.
23. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In *Information Systems Evolution*, volume 72, pages 60–75, 2010.
24. Z. Zhou, Y. Wang, and L. Li. Process Mining Based Modeling and Analysis of Workflows in Clinical Care - A Case Study in a Chicago Outpatient Clinic. In *ICNSC 2014*, pages 590–595, 2014.