DB-XES: Enabling Process Discovery in the Large

Alifah Syamsiyah, Boudewijn Dongen, Wil Aalst

DB-XES: Enabling Process Discovery in the Large

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

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

Abstract. Dealing with the abundance of event data is one of the main process discovery challenges. Current process discovery techniques are able to efficiently handle imported event log files that fit in the computer's memory. Once data files get bigger, scalability quickly drops since the speed required to access the data becomes a limiting factor. This paper proposes a new technique based on relational database technology as a solution for scalable process discovery. A relational database is used both for storing event data (i.e. we move the location of the data) and for pre-processing the event data (i.e. we move some computations from analysis-time to insertion-time). To this end, we first introduce DB-XES as a database schema which resembles the standard XES structure, we provide a transparent way to access event data stored in DB-XES, and we show how this greatly improves on the memory requirements of the state-of-the-art process discovery techniques. Secondly, we show how to move the computation of intermediate data structures to the database engine, to reduce the time required during process discovery. The work presented in this paper is implemented in ProM tool, and a range of experiments demonstrates the feasibility of our approach.

Keywords: Process discovery · Process mining · Big event data · Relational database

1 Introduction

Process mining is a research discipline that sits between machine learning and data mining on the one hand and process modeling and analysis on the other hand. The goal of process mining is to turn event data into insights and actions in order to improve processes [25]. One of the main perspectives offered by process mining is process discovery, a technique that takes an event log and produces a model without using any a-priori information. Given the abundance of event data, the challenge is to enable *process discovery in the large*. Any sampling technique would lead to statistically valid results on mainstream behavior, but would not lead to insights into the exceptional behavior, which is typically the goal of process mining.

Process mining has been successfully implemented in dozens case studies, ranging from healthcare [15,28,35], industry [16,18,20], finance [9,10], etc. Suppose that managers of an insurance company are interested to discover their

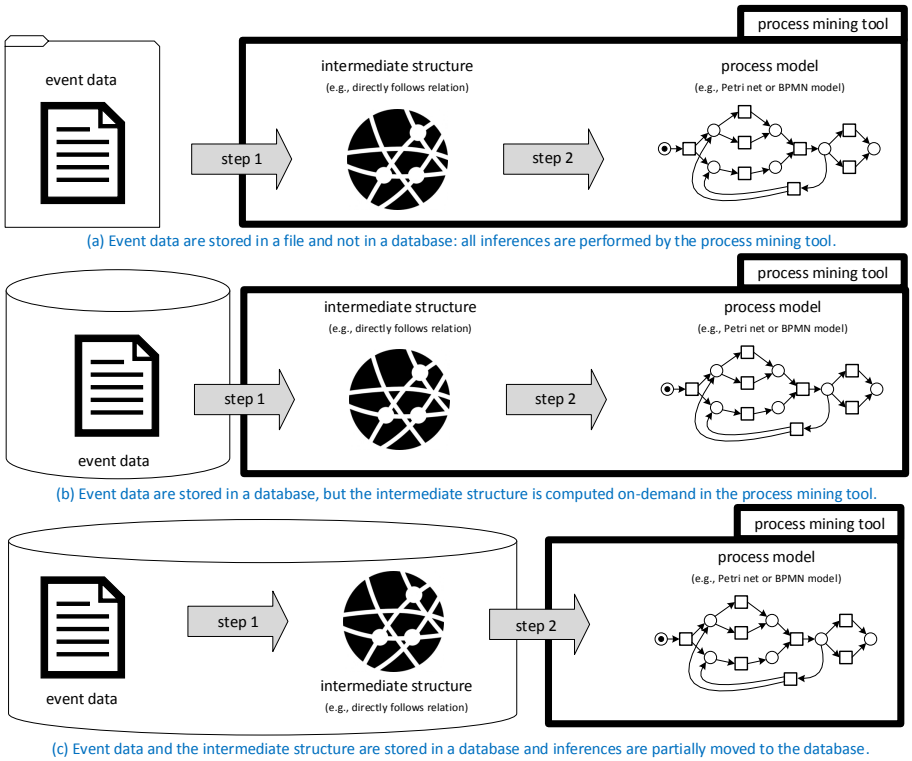

Fig. 1: Three different settings in process discovery

business process models representing a claim handling scheme. The claim handling system is always running while producing event data which are valuable for monitoring and evaluation. Based on such growing event data, the company's managers discover the process models and report the analysis results to their director every month. Note that in order to perform a comprehensive evaluation, the managers need to incorporate event data from previous months when discovering a process model. This scenario shows a usage example of recurrent process discovery based on event data that grows over time.

In the traditional setting of process discovery, event data is read from an event log file and a process model describing the recorded behavior is produced, as depicted in Figure 1(a). In between, there is a so-called *intermediate structure*, which is an abstraction of event data in a structured way, e.g. the directly follows relation, a prefix-automaton, etc. To build such an intermediate structure, process mining tools load the event log in memory and build the intermediate structure in the tool, hence the analysis is bound by the memory needed to store both the event log and the intermediate structure in memory. The time needed for the analysis includes the time needed to convert the log to the intermediate structure. Furthermore, in the context of recurrent process discovery, one

needs to reload and recompute the previous data since the previous results are discarded from memory when process mining tools are terminated.

To increase the scalability, relational databases have been proposed for storing event data [31], as depicted in Figure 1(b), i.e. the event log file is replaced by a database. In [31] a database schema was introduced to store event data and experiments showed the reduction in memory use. A connection is established from the database to process mining tools to access the event data *on demand* using the standard interfaces for dealing with event logs, i.e. OpenXES [7]. Since no longer the entire event log is to be read in memory, the memory consumption of the process mining analysis will be shown to be reduced significantly as now only the intermediate structure needs to be stored. However, this memory reduction comes at a cost of analysis time since access to the database is several orders of magnitude slower than access to an in-memory event log while building the intermediate structure for further analysis.

Therefore, we present a third solution, called DB-XES, where we not only move the location of the event data, but also the location of such intermediate structures. In order to do so, we move the computation of intermediate structures from analysis time to insertion time, as depicted in Figure 1(c). In other words, each intermediate structure is kept up-to-date for each insertion of a new event of a trace in the database. Moreover, both event data and intermediate structures are kept in a persistent storage, hence there is no need to reload and recompute the previous data for recurrent analysis.

In this paper we present the general idea and a concrete instantiation using intermediate structures of the state-of-the-art process discovery techniques. We consider both procedural and declarative paradigms in process discovery as to demonstrate a broad usage of the proposed technique. Finally, we show that the proposed solution saves both memory and time during process analysis.

The remainder of this paper is organized as follows. In Section 2, we discuss some related work. In Section 3, we present the database schema for DB-XES. In Section 4, we extend DB-XES with the notion of intermediate structures. In Section 5 and Section 6 we show how two well-known intermediate structures can be computed inside the database. Then, in Section 7, we present the implementation of the idea as ProM plug-ins. In Section 8 we present experiments which show significant performance gains. Finally, we conclude and discuss the future work in Section 9.

2 Related Work

One of the first tools to extract event data from a database was XESame [33]. In XESame users can interactively select data from the database and then match it with XES elements. However, the database is only considered as a storage place of data as no direct access to the database is provided.

Similar to XESame, in [2] a technique is presented where data stored in databases is serialized into an XES file. The data is accessed with the help of two ontologies, namely a *domain ontology* and an *event ontology*. Besides that,

the work also provided on-demand access to the data in the database using query unfolding and rewriting techniques in Ontology Based Data Access [17]. However, the performance issues make this approach unsuitable for large databases.

Some commercial tools, such as Celonis[1] and Minit[2], also incorporate features to extract event data from a database. The extraction can be done extremely fast, however, its architecture has several downsides. First, it is not generic since it requires a transformation to a very specific schema, e.g. a table containing information about case identifier, activity name, and timestamp. Second, it cannot handle huge event data which exceed computer's memory due to the fact that the transformation is done inside the memory. Moreover, since no direct access to the database is provided, some updates in the database will lead to restarting of the whole process in order to get the desired model.

Building on the idea of direct access to the database, in [31], RXES was introduced before as the relational representation of XES and it is was shown that RXES uses less memory compared to the file-based OpenXES and MapDB XES Lite implementations [14]. However, its application to a real process mining algorithm was not investigated and the time-performance analysis was not included.

In [34], the performance of multidimensional process mining (MPM) is improved using relational databases techniques. It presented the underlying relational concepts of PMCube, a data-warehouse-based approach for MPM. It introduced generic query patterns which map OLAP queries to SQL to push the operations to the database management systems. This way, MPM may benefit from the comprehensive optimization techniques provided by state-of-the-art database management systems. The experiments reported in the paper showed that PMCube provides a significantly better performance than PMC, the state-of-the-art implementation of the Process Cubes approach.

The use of database in process mining gives significance not only to the procedural process mining, but also declarative process mining. The work in [21] introduced an SQL-based declarative process mining approach that analyses event log data stored in relational databases. It deals with existing issues in declarative process mining, namely the performance issues and expressiveness limitation to a specific set of constraints. By leveraging database performance technology, the mining procedure in SQLMiner can be done fast. Furthermore, SQL queries provide flexibility in writing constraints and it can be customized easily to cover process perspective beyond control flow. However, none of these techniques handles live event data, the focus is often on static data that has been imported in a database.

Apart from using databases, some other techniques for handling big data in process mining have been proposed [1, 19, 26], two of them are decomposing event logs [24] and streaming process mining [8, 13, 32]. In decomposition, a large process mining problem is broken down into smaller problems focusing on a restricted set of activities. Process mining techniques are applied separately in

[1] http://www.celonis.de/en/
[2] http://www.minitlabs.com/

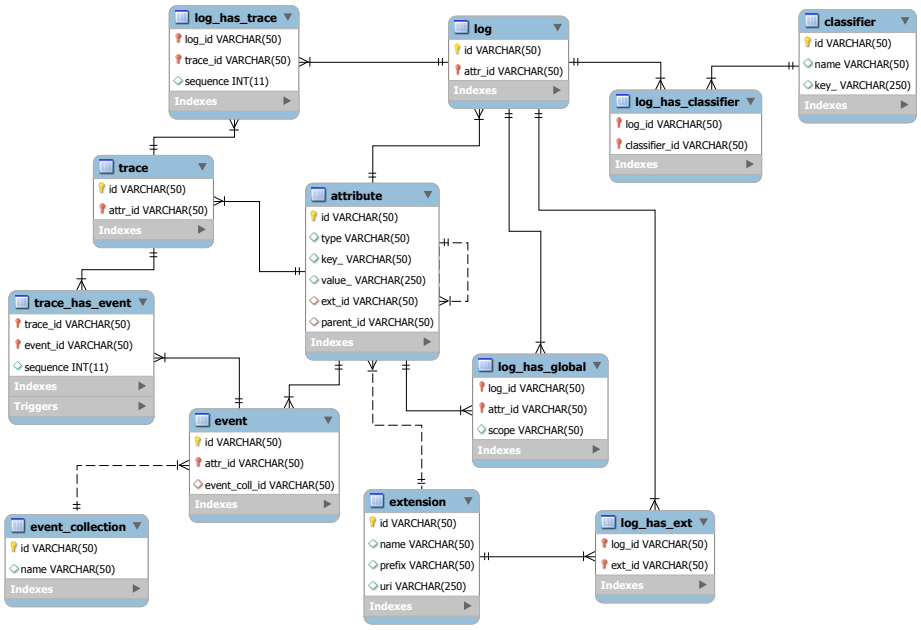

Fig. 2: DB-XES basic schema

each small problem which then they are combined to get an overall result. This approach deals with exponential complexity in the number of activities of most process mining algorithms [23]. Whereas in streaming process mining, it provides online-fashioned process mining where the event data is freshly produced, i.e. it does not restrict to only process the historical data as in traditional process mining. Both approaches however require severe changes to the algorithms used for analysis and they are therefore not directly applicable to existing process mining techniques.

3 DB-XES as Event Data Storage

In the field of process mining, event logs are typically considered to be structured according to the XES standard [7]. Based on this standard, we create a relational representation for event logs, which we called *DB-XES*. We select relational databases rather than any other type of databases, e.g. NoSQL [22], because of the need to be able to slice and dice data in different ways. An e-commerce system, for example, may need to be analyzed using many views. One view can be defined based on customer order, other view may also be defined based on delivery, etc. Some NoSQL databases, such as key-value store databases, document databases, or column-oriented databases, are suitable for the data which can be aggregated, but have difficulties supporting multiple perspectives at the same time. Besides, we select relational databases because of the need to have

a trigger for updating intermediate structure tables automatically. Relational databases are more mature than NoSQL databases with respect to database features, including trigger operations.

Figure 2 shows the basic database schema of DB-XES. The XES main elements are represented in tables *log, trace, event,* and *attribute*. The relation between these elements are stored in tables *log_has_trace* and *trace_has_event*. Furthermore, classifier and extension information related to a log can be accessed through tables *log_has_classifier* and *log_has_extension*. Global attributes are maintained in the table *log_has_global*. In order to store the source of event data, we introduce the *event collection* table.

OpenXES is a Java-based reference implementation of the XES standard for storing and managing event log data [7]. OpenXES is a collection of interfaces and corresponding implementations tailored towards accessing XES files. In consequence of moving event data from XES files to DB-XES, we need to implement some Java classes in OpenXES. Having the new version of OpenXES, it allows for any process mining techniques capable of handling OpenXES data to be used on DB-XES data. The implementation is distributed within the *DBXes* package in ProM (`https://svn.win.tue.nl/repos/prom/Packages/DBXes/Trunk/`).

The general idea is to create SQL queries to get the event data for instantiating the Java objects. Access to the event data in the database is defined for each element of XES, therefore we provide *on-demand access*. We define a log, a trace, and an event based on a string identifier and an instance of class *Connection* in Java. The identifier is retrieved from a value under column *id* in *log, trace,* and *event* table respectively. Whereas the instance of class *Connection* should refer to the database where we store the event data. Upon initialization of the database connection, the list of available identifiers is retrieved from the database and stored in memory using global variables.

4 Extending DB-XES with Intermediate Structures

In the analysis, process mining rarely uses event data itself, rather it processes an abstraction of event data called an *intermediate structure*. This section discusses the extension of DB-XES with intermediate structures. First, we briefly explain about several types of intermediate structures in process mining, then we present the highly used intermediate structures in procedural and declarative process discoveries that we implemented in DB-XES.

There are many existing intermediate structures in process mining, such as the eventually follows relation, no co-occurrence relation [3,5], handover of work relation [27], and prefix-closed languages in region theory [29]. Each intermediate structure has its own functions and characteristics. Some intermediate structures are robust to filtering, hence we may get different views on the processes by filtering the event data without recalculation of the intermediate structure like eventually follows relation, but some require full recomputation [25]. Mostly intermediate structures can be computed by reading the event data in a single pass over the events, but some are more complex to be computed. In general the

size of intermediate structure is much smaller than the size of the log [3, 5, 27], but some intermediate structures are bigger than the log [29]. In the following we briefly introduce some examples of intermediate structures.

- The directly follows relation ($a > b$) contains information that *a is directly followed by b in the context of a trace*. This relation is not robust to filtering. Once filtering happens, the relation must be recalculated. Suppose that a is directly followed by b, i.e. $a > b$, and b is directly followed by c, i.e. $b > c$. If we filter b, now a is directly followed by c, hence a new relation $a > c$ holds.
- The eventually follows relation ($V(a, b)$) is the transitive closure of the directly follows relation: *a is followed by b somewhere in the trace*. Suppose that a is eventually followed by b, i.e. $V(a, b)$, and a is eventually followed by c, i.e. $V(a, c)$. If we filter b, a is still followed by c somewhere in the trace, i.e. $V(a, c)$ still holds. Therefore, eventually follows relation is robust to filtering.
- The no co-occurrence relation ($R(a, b)$) counts *the occurrences of a with no co-occurring b in the trace*. For example, a occurs four times with no co-occurring b, i.e. $R(a, b) = 4$, and a occurs three times with no co-occurring c, i.e. $R(a, c) = 3$. If we filter b, it does not effect the occurrence of a with no c, i.e. $R(a, c) = 3$ still holds. Therefore, no co-occurrence relation is robust to filtering.
- The handover of work relation between individual a and b ($H(a, b)$) exists if *there are two subsequent activities where the first is completed by a and the second by b*. This is also an example of non-robust intermediate structure for filtering. Imagine we have $H(a, b)$ and $H(b, c)$. When b is filtered, a directly handed over to c, hence $H(a, c)$ must be deduced. This indicates the whole relations need to be recalculated.
- The Integer Linear Programming (ILP) Miner uses language-based theory of regions in its discovery. The regions are produced from a prefix-closed language which is considered as the intermediate structure. As an example, we have log $L = \{\langle a, b, c\rangle, \langle a, d, e\rangle\}$. The prefix-closed language of L is $\mathcal{L} = \{\epsilon, \langle a\rangle, \langle a, b\rangle, \langle a, d\rangle, \langle a, b, c\rangle, \langle a, d, e\rangle\}$. It is clear that $\mathcal{L}$ is bigger than L. The prefix-closed language in region theory is one of the intermediate structures whose size is bigger than the log size. It is not robust to filtering. Based on the event log L above, suppose that b is filtered, then $\langle a, c\rangle$ must be added to $\mathcal{L}$.

While many intermediate structures can be identified when studying process mining techniques, we currently focus on the *Directly Follows Relation (DFR)* as a representative of procedural process discoveries and MINERful relations [6] as a representative of declarative process discoveries. The DFR is used in many process mining algorithms, including the most widely used process discovery techniques, such as Inductive Miner [11, 12]. The MINERful has demonstrated the best scalability with respect to the input size compared to the other declarative discovery techniques [6]. In the following we discuss how DB-XES is extended with DFR and MINERful tables.

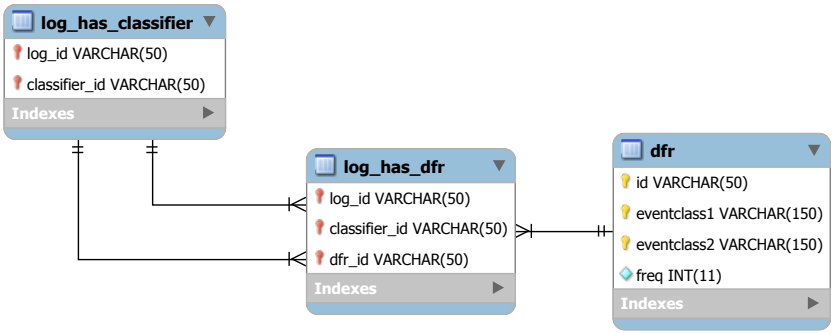

Fig. 3: DFR in DB-XES schema

4.1 The DFR Intermediate Structure in DB-XES

Directly Follows Relation (DFR) contains information about *the frequency with which one event class directly follows another event class in the context of a trace*. Following the definition in [25], DFR is defined as follows.

Definition 1 (Event log). *Let Υ be the universe of events and $E \subseteq \Upsilon$ be a collection of events. An event log $L \subseteq E^*$ is a set of event sequences (called traces) such that each event appears precisely once in precisely one trace.*

Definition 2 (Event attributes and classifiers). *Let Υ be the universe of events, C be the universe of cases, N be the universe of attribute names, and A be the universe of activities.*

- *For any event $e \in \Upsilon$ and name $a \in A$: $\#_a(e)$ is the value of attribute a for event e. $\#_a(e) = \perp$ if there is no value.*
- *For any event $e \in \Upsilon$ and $\{act, cs, tm\} \subseteq N$, we define $\#_{act} : \Upsilon \to A$ a function labeling each event with an activity, $\#_{cs} : \Upsilon \to C$ a function labeling each event with a case, and $\#_{tm} : \Upsilon \to \mathbb{R}$ a function labeling each event with a timestamp.*
- *Any subset $N' \subseteq \{a_1, a_2, ..., a_n\} \subseteq N$ is a classifier, i.e., an ordered set of attributes. We define: $\#_{N'}(e) = (\#_{a_1}(e), \#_{a_2}(e), ..., \#_{a_n}(e))$.*
- *In the context of an event log there is a default classifier $DC \subseteq N$ for which we define the shorthand of event class $\underline{e} = \#_{DC}(e)$.*

Definition 3 (Directly Follows Relation (DFR)). *Let $L \subseteq E^*$ be an event log. x is directly followed by y, denoted $x > y$, if and only if there is a trace $\sigma = \langle e_1, e_2, ..., e_n \rangle \in L$ and $1 \leqslant i < n$ such that $\underline{e_i} = x$ and $\underline{e_{i+1}} = y$.*

Translated to DB-XES, table *dfr* consists of three important columns next to the *id* of the table, namely $eventclass_1$ which indicates the first event class in directly follows relation, $eventclass_2$ for the second event class, and *freq* which indicates how often an event class is directly followed by another event class. Figure 3 shows the position of table *dfr* in DB-XES. As DFR is defined on the

event classes based on a classifier, every instance in table *dfr* is linked to an instance of table *classifier* in the log.

Definition 4 (Table dfr). *Let $L \subseteq E^*$ be an event log, $X = \{\underline{e} \mid e \in E\}$ is the set of event classes.* $\mathrm{dfr} \in X \times X \nrightarrow \mathbb{N}$ *where:*

- $dom(\mathrm{dfr}) = \{(x, y) \in X \times X \mid x > y\}$
- $\mathrm{dfr}(x, y) = \sum_{\langle e_1,...,e_n\rangle \in L} |\{i \in \{1, ..., n-1\} \mid \underline{e_i} = x \wedge \underline{e_{i+1}} = y\}|$

As mentioned before, the design choice to incorporate DFR as the intermediate structure is due to fact that DFR is used in the state-of-the-art procedural process discovery algorithm. In the next section, we discuss how to extend DB-XES with intermediate structures of the state-of-the-art declarative process discovery algorithm, namely MINERful relations.

4.2 The MINERful Intermediate Structures in DB-XES

During abstraction phase, 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 5 (MINERful relations [4]).
Let L be an event log over $E \subseteq \Upsilon$. The following relations are defined for MINERful:

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

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

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

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

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

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

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

$\not\rightsquigarrow_L : A \times A \to \mathbb{N}$ *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.

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

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

Similar to DFR, MINERful relations are translated to DB-XES tables which consists of column identifier, activity name(s), and frequency.

5 DFR Pre-Computation in DB-XES

Typically, process mining algorithms build an intermediate structure in memory while going through the event log in a single pass (as depicted in Figure 1(a)). However, this approach will not be feasible when handling huge event log whose size exceeds the computer memory. Moving the location of the event data from a file to a database as depicted in Figure 1(b) increases the scalability of process mining as the computer memory no longer needs to contain the event data. However, the *ping-pong* communication generated when iterating over the log in the database from a process mining tool is time consuming. Therefore, in this section, we show how DFR is pre-computed in DB-XES (Figure 1(c)). Particularly, we show how common processing tasks can be moved both in time and location, i.e. we show how to *store intermediate structures in DB-XES* and we show how these structures can be *updated while inserting the data* rather than when doing the process mining task.

As mentioned in Section 4, the table *dfr* in Figure 3 is the table in DB-XES which stores DFR values, furthermore, the table *log_has_dfr* stores the context in which the DFR exists, i.e. it links the DFR values to a specific log and classifier combination. The *dfr* table is responsive to update operations, particularly when users insert new events to the log. In the following we discuss how the *dfr* table is created and updated in DB-XES.

5.1 Creating Table *dfr* in DB-XES

Suppose that there exist two entries in the *trace_has_event* table with trace id σ, event id's e_i and e_{i+1} and sequence's i and $i+1$. The first event e_i is linked to an attribute α with *value* a and the second event is linked to an attribute α with *value* b while the log has a classifier based on attribute α. In DB-XES, we store the frequency of each pair $a > b$ in the database rather than letting the discovery algorithm build it on-demand and in-memory. In other words, the directly follows relation is precomputed and the values can be retrieved directly by a process mining algorithm when needed.

To create table *dfr*, we run three SQL queries. The first query is to obtain pairs of directly follows relations. For instance, if an event class a is directly followed by an event class b and this happens 100 times in the log, then there will be a row in table *dfr* with value *(dfr_1, a, b, 100)*, assuming the id is dfr_1. Furthermore, the second and third queries are to extract start and end event classes. We create an artificial start ($\top$) and end ($\bot$) event for each process instance. For example, if there are 200 cases where a happens as the start event class, there will be a row in *dfr* with values *(dfr_1, $\top$, a, 200)*. Similarly, if b is the end event class for 150 cases, there will be a row in *dfr* with values *(dfr_1, b, $\bot$, 150)*.

Technically, the SQL query contains big joins between tables *trace_has_event, event, attribute, log_has_trace, log_has_classifier,* and *classifier*. Such joins are needed to get pairs of event classes whose events belong to the same trace in the same log which has some classifiers. The SQL query mentioned below is a simplified query to obtain pairs of directly follows relations. To improve understandability, we use placeholders (< ... >) to abstract some details. Basically they are trivial join conditions or selection conditions to interesting columns.

```
1  SELECT id, eventClass1, eventClass2, count(*) as freq
2  FROM (
3         SELECT <...>
4         FROM (
5                 SELECT <...>
6                 FROM trace_has_event as t1
7                      INNER JOIN trace_has_event as t2
8                      ON t1.trace_id = t2.trace_id
9                 /* Consecutive events have subsequent
10                   sequence numbers in the trace */
11                WHERE t1.sequence = t2.sequence - 1
12               ) as pair_of_events,
13              attribute as a1, attribute as a2,
14              event as event1, event as event2,
15              log_has_trace, log_has_classifier, classifier
16        WHERE <...>
17        GROUP BY log_id, classifier_id,
18                 event1.id, event2.id
19     ) as pair_of_eventclasses
20 GROUP BY id, eventClass1, eventClass2
```

We start with a self join in table *trace_has_event* (line 6-8) to get pairs of two events which belong to the same trace. Then we filter to pairs whose events happen consecutively, i.e. the sequence of an event is preceded by the other (line 11). Note that this sequence attribute requires events are loaded into database in a chronological order. The next step is obtaining the attribute values of these events. The attribute values are grouped based on the classifier in the log (line 17-18). This grouping is essential if the classifier is built from a combination of several attributes, for example a classifier based on the activity name and

lifecycle. After grouping, we get a multiset of pairs of event classes. Finally, the same pairs are grouped and counted to obtain the frequency of how often they appeared in the log (line 1, 20). The next SQL query shows how to obtain start event classes from DB-XES.

```
1  SELECT id, 'start', startEventClass, count(*) as freq
2  FROM (
3         SELECT <...>
4         FROM trace_has_event, event, attribute,
5              log_has_trace, log_has_classifier, classifier
6         /* First events in a trace get sequence 0 */
7         WHERE sequence = 0 AND <...>
8         GROUP BY log_id, classifier_id, event_id
9       ) as pair_of_eventclasses
10 GROUP BY id, startEventClass
```

The SQL query to get start event classes is simpler. Start event classes are indicated by their sequence is equal to zero (line 7). In the case of end event classes, they are indicated by their sequence is equal to the length of the trace. We put a constant 'start' as an artificial start ($\top$) to fulfill condition as a pair in directly follows relations (line 1). Furthermore, the rests of the query are identical to the SQL query for obtaining pair of event classes as mentioned before.

5.2 Updating Table *dfr* in DB-XES

Rows in table *dfr* are automatically updated whenever users insert a new event through a trigger operation on table *trace_has_event* which is aware of an insert command. Here we consider two scenarios: (1) a newly inserted event belongs to a new trace in a log for which a dfr table exists and (2) a newly inserted event belongs to an existing trace in such a log. We assume such insertion is well-ordered, i.e. an event is not inserted at an arbitrary position.

Suppose that we have a very small log $L = [\langle a, b \rangle]$, where we assume a and b refer to the event class of the two events in L determined by a classifier cl for which an entry *(L, cl, dfr_1)* exists in the *log_has_dfr* table. This log only contains one trace (say σ_1) with two events that correspond to two event classes, namely a and b. If we add to L a new event with a new event class c to a new trace different from σ_1 then such an event is considered as in the first scenario. However, if we add c to σ_1 then it is considered as the second scenario.

In the first scenario, we update the start and end frequency of the inserted event type. In our example above, the rows in table *dfr* containing *(dfr_1, $\top$, c, f)* and *(dfr_1, c, $\bot$, f)* will be updated as *(dfr_1, $\top$, c, f + 1)* and *(dfr_1, c, $\bot$, f + 1)* with f is the frequency value. If there is no such rows, *(dfr_1, $\top$, c, 1)* and *(dfr_1, c, $\bot$, 1)* will be inserted.

In the second scenario, we update the end frequency of the last event class before the newly inserted event class, and add the frequency of the pair of those two. Referring to our example, row *(dfr_1, b, $\bot$, f)* is updated to *(dfr_1, b, $\bot$, f - 1)*. If there exists row *(dfr_1, c, $\bot$, f)*, it is updated to *(dfr_1, c, $\bot$, f + 1)*,

otherwise *(dfr$_1$, c, $\perp$, 1)* is inserted. Furthermore, if *(dfr$_1$, b, c, f)* exists in table *dfr*, it is updated as *(dfr$_1$, b, c, f + 1)*, otherwise *(dfr$_1$, b, c, 1)* is inserted.

By storing the intermediate structure in the database and updating this structure when events are inserted, we move a significant amount of computation time to the database rather than to the process analysis tool. This allows for faster analysis with virtually no limits on the size of the event log as we show in Section 8.

6 MINERful Relations Pre-Computation in DB-XES

In the previous section we have seen how DFR are created and kept up-to-date in DB-XES. Keeping that relation live under updates is rather trivial. Most MINERful relations in Definition 5 however do not allow for a simple update strategy. In this section we discuss pre-computation of MINERful relations in DB-XES, particularly the update technique. We leave out the details of the creation technique of MINERful relations since they can be adopted easily from the DFR creation technique in Section 5.1.

6.1 Updating Table MINERful in DB-XES

First of all, 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 MINERful 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 $c \in C$. $\not\vdash^c_L : A \times A \to \mathbb{N}$ is a controller function such that for all $a, b \in A$ 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 $a \in A$ with no following $b \in A$ 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 5. 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 MINERful relations in Definition 5 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.

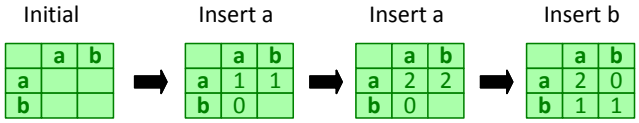

Fig. 4: An example of updating controller function $\not\vdash^c$. Events (which are represented by their activity names) in trace $\sigma_c = \langle a, a, b \rangle$ are inserted one by one and in each insertion values under $\not\vdash^c$ are updated, assuming $A = \{a, b\}$.

Lemma 1 (Updating controller function is possible).
Let $E \subseteq \Upsilon$ be a set of events and L a log over E. Let $e \in \Upsilon \setminus E$ 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. 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 4 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 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 directly follows relation as the latter only has to consider the last label in the trace of the new event.

Lemma 2 (Updating controller function is sufficient to update MINERful relations).
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.

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

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

$$\not\rightarrow_{L'}(a,b) = \not\rightarrow_L(a,b) + \begin{cases} -\not\rightarrow^c_L(a,b) + \not\rightarrow^c_{L'}(a,b) & \text{if } a \neq b, \\ 0 & \text{otherwise} \end{cases}$$

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

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

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

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

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.

$\not\rightarrow(a,b)$ 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 $a \neq b$.

$\not\leftarrow(a,b)$ 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.

$\not\twoheadrightarrow(a,b)$ 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.

$\leftsquigarrow(a,b)$ 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 1 (Updating all MINERful relations is possible).
Relations $\#, \not\vdash, \not\dashv, \not\dashv\vdash, \not\rightarrow$, *and* $\not\leftarrow$ *can be kept live under insertion of new events.*

Proof. 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 $\#, \not\vdash, \not\dashv, \not\dashv\vdash, \not\rightarrow$, and $\not\leftarrow$ (Lemma 2). Therefore, it is possible to keep those intermediate structures up-to-date in each insertion of a new event. ■

In Theorem 1 we have proven that updating all MINERful relations is possible, hence in DB-XES we can kept these relations live under insertion of new events. In this way, we save a significant amount of computation time since MINERful relations are precomputed inside the database rather than inside the process mining tool.

7 Implementation

We implemented the proposed technique as a ProM[3] plug-in which integrates DB-XES with the state-of-the-art process discovery algorithm, namely the Inductive Miner. There are two plug-in variants (Figure 5): (1) *Database-Incremental Inductive Miner (DIIM)* for discovery using user-defined classifiers, and (2) *Database-Incremental Inductive Miner with Concept Name (DIIMcn)* for discovery using the standard *concept:name* classifier. The implementation is distributed within *DatabaseInductiveMiner* package (`https://svn.win.tue.nl/repos/prom/Packages/DatabaseInductiveMiner/Trunk/`).

Figure 6(a) shows the first interface of DIIM/DIIMcn. There are some configurations required for establishing a connection between DB-XES and ProM, including a username of database account, password, server, database name, and a log identifier (an identifier in table *log* indicating the process we want to discover). Then, based on these configurations, the plug-in lists all classifiers linked to the log (Figure 6(b)). Users have to pick one classifier for categorizing events. Furthermore, users may choose to create the intermediate structure (DFR) from scratch or use pre-computed DFR which is already exist in DB-XES. The former option is needed if there is no available DFR in DB-XES. It applies, for example, to the old data that is stored before the trigger to update DFR is activated.

After all configurations are set, the plug-in shows the discovered process model based on the Inductive Miner algorithm. We utilize existing algorithm which does not aware whether the event data is taken from database or normal XES file. Figure 7 depicts a simple process model discovered by DIIM/DIIMcn based on event log $L = [\langle a, a, b, a, c, a\rangle, \langle a, a, b, a, c, a, d\rangle]$.

[3] `http://www.promtools.org/`

Fig. 5: Two plug-in variants: DIIM and DIIMcn

(a) Configuration for database connection (b) Configuration for classifier and DFR

Fig. 6: DIIM/DIIMcn configurations to connect DB-XES with Inductive Miner in ProM

Furthermore, we implemented DB-XES with MINERful relations as a ProM plug-in called *Database-Incremental Declare Miner (DIDM)* which is distributed within *MixedParadigm* package (`https://svn.win.tue.nl/repos/prom/Packages/MixedParadigm/Trunk/`). 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.*

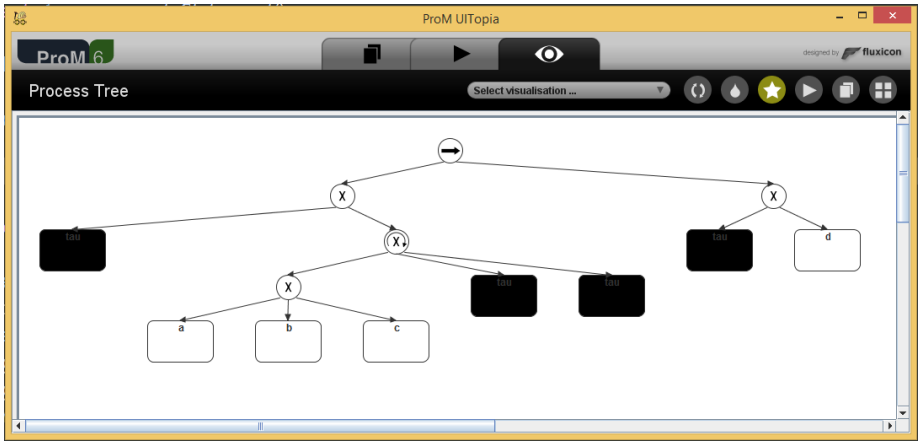

Fig. 7: The discovered process model in DIIM/DIIMcn

8 Experiments

In this section we show the influence of moving both the event data and the directly follows table to the database on the memory use and time consumption of the Inductive Miner. Next to the traditional in-memory processing of event logs (Figure 1(a)), we consider two scenarios in DB-XES: (1) *DB-XES without DFR* where the intermediate result is computed during the discovery (Figure 1(b)) and (2) *DB-XES with DFR* where the intermediate result is pre-computed in the database (Figure 1(c)). We show that the latter provide scalability with respect to data size and even improves time spent on actual analysis.

Furthermore, we dive into each phase of the proposed technique in more detail. We look into the end-to-end process, starting from inserting event data into DB-XES, updating the intermediate structures, and finally mining a process model both in procedural and declarative ways. In this experiment, we apply DIIM and DIDM in a live scenario and compare it to traditional Inductive Miner and MINERful.

In the following we first show the experimental results of memory use and CPU time and then the experimental results of the end-to-end process discovery using DB-XES. Both experiments were executed on the machine with processor Intel(R) Core(TM) i7-4700MQ, 16GB of RAM, and an external MySQL server version 5.7.12.

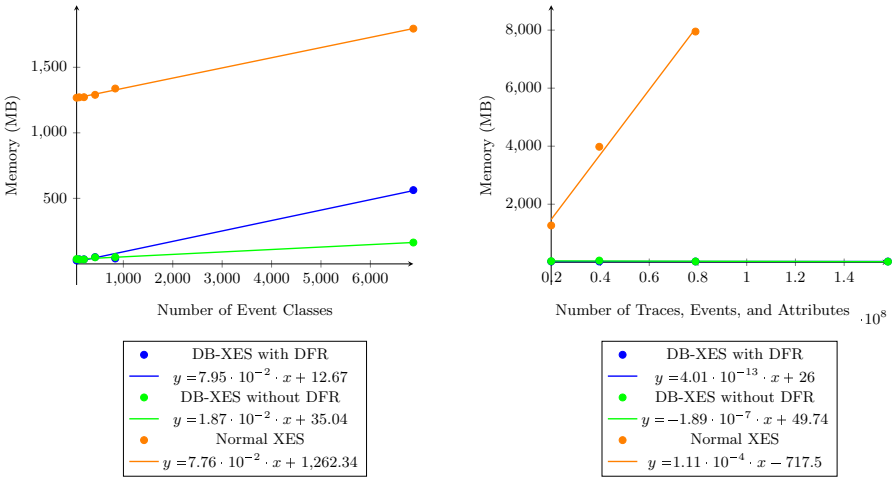

Fig. 8: From left to right: memory use of the Inductive Miner in: (a) logs with extended event classes and (b) logs with extended traces, events, and attributes

8.1 Memory Use and CPU Time

As the basis for the experiments, we use an event log from a real company which contains 29,640 traces, 2,453,386 events, 54 different event classes and 17,262,635 attributes. Then we extend this log in two dimensions, i.e. we increase (1) the number of event classes and (2) the number of traces, events and attributes. We extend the log by inserting copies of the original event log data with some modifications in the identifier, task name, and timestamp. In both cases, we keep the other dimension fixed in order to get a clear picture of the influence of each dimension separately on both memory use and CPU time.

In Figure 8(a), we show the influence of increasing the number of event classes on the memory use of the Inductive Miner. The Inductive Miner makes a linear pass over the event log in order to build an object storing the direct succession relation in memory. In theory, the direct succession relation is quadratic in the number of event classes, but as only actual pairs of event classes with more than one occurrence are stored and the relation is sparse, the memory consumption scales linearly in the number of event classes as shown by the trendlines. It is clear that the memory use of DB-XES is consistently lower than XES. This is easily explained as there is no need to store the event log in memory. The fact that DB-XES with DFR uses more memory than DB-XES without DFR is due to the memory overhead of querying the database for the entire DFR table at once. Note that the DFR table grows from 304 pairs (with 54 distinct event classes) to 17,819 pairs (with 6,870 distinct event classes).

In Figure 8(b), we present the influence of increasing the number of events, traces and attributes while keeping the number of event classes constant. In this case, normal XES quickly uses more memory than the machine has while both

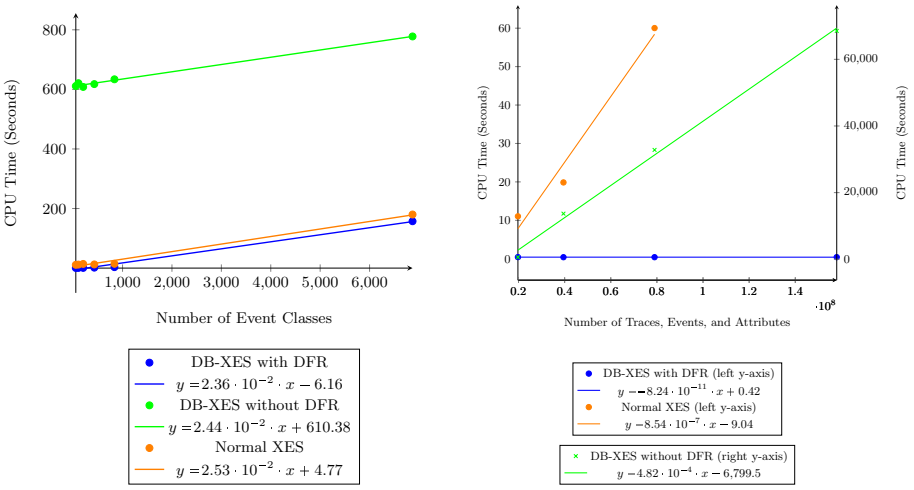

Fig. 9: From left to right: CPU time of the Inductive Miner in: (a) logs with extended event classes and (b) logs with extended traces, events, and attributes

DB-XES implementations show no increase in memory use with growing data and the overall memory use is less than 50 MB. This is expected as the memory consumption of the Inductive Miner varies with the number of event classes only, i.e. the higher frequency values in the *dfr* table do not influence the memory use.

We also investigated the influence of accessing the database to the CPU time needed by the analysis, i.e. we measure the time spent to run the Inductive Miner. In Figure 9(a), we show the influence of the number of event classes on the CPU time. When switching from XES files to DB-XES without DFR, the time needed to do the analysis increases considerably. This is easily explained by the overhead introduced in Java by initiating the query every time to access an event. However, when using DB-XES with DFR, the time needed by the Inductive Miner decreases, i.e. it is faster to obtain the *dfr* table from the database than to compute it in memory.

This effect is even greater when we increase the number of traces, events and attributes rather than the number of event classes as shown in Figure 9(b). DB-XES with DFR shows a constant CPU time use, while normal XES shows a steep linear increase in time use before running out of memory. DB-XES without DFR also requires linear time, but is several orders of magnitude slower (DB-XES without DFR is drawn against the right-hand side axis).

In this section, we have proven that the use of relational databases in process mining, i.e DB-XES, provide scalability in terms of memory use. However, accessing DB-XES directly by retrieving event data elements on demand and computing intermediate structures in ProM is expensive in terms of processing time. Therefore, we presented DB-XES with DFR where we moved the com-

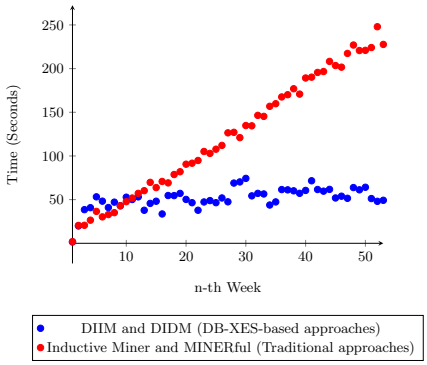

Fig. 10: The comparison of recurrent process discovery using DIIM and DIDM vs traditional Inductive Miner and MINERful

putation of the intermediate structure to the database. This solution provides scalability in both memory and time.

8.2 End-to-end Process Discovery using DB-XES

In this section, we show the end-to-end process discovery using DB-XES. Using two DB-XES-based implementations, namely DIIM and DIDM, we observe the time for inserting events into DB-XES, updating the intermediate structures, until mining the process model. Then we compare these techniques with traditional techniques using Inductive Miner and MINERful.

We used a real dataset from BPI Challenge 2017 [30] for the experiment. This dataset relates 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 are interested to have some weekly progress reports. In the end of each week, we discover both procedural and declarative process models. These weekly discoveries considers a collective data since the beginning of the year (since 2016).

In DB-XES-based approach, 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, thus keeping the relations live under updates. In traditional approach, 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 10 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 declarative process models. The blue dots are the experiment

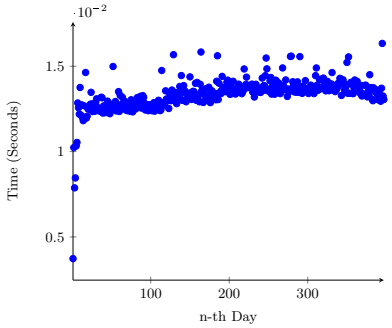

Fig. 11: Average update time per event

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 10, after the first two months, our DB-XES-based techniques became faster, even when considering the time needed to insert events in the DB-XES, a process that is typically executed in real time and without the business analyst being present. In the traditional settings, the time to perform the process mining task 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 (loading the data into process mining tool and doing the abstraction of the intermediate structures) scales linearly in the number of events, whereas the mining phase scales in the number of activities. The latter is considerably smaller than the former in most practical cases as well as in this example. In contrast, our DB-XES approaches are more stable over time as the update phase only depends on the number of *newly inserted* events and both the retrieval and mining phases 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 11. The x-axis represents the n-th day, while the y-axis represents the update time per event. As shown from the Figure 11, 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 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.

9 Conclusion and Future Work

This paper focuses on the issue of scalability in terms of both memory use and CPU use in process discovery. We introduce a relational database schema called DB-XES to store event data and we show how intermediate structures can be stored in the same database and be kept up-to-date when inserting new events into the database. We use Directly Follow Relations (DFR) and MINERful relations as examples of intermediate structures, but the work trivially extends to other intermediate structures as long as they can be kept up-to-date during insertion of event data in the database. 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 experiments on real-life data we show that storing event data in DB-XES not only leads to a significant reduction in memory use of the process mining tool, but can even speed up the analysis if the pre-processing is done in the right way in the database upon insertion of the event data. Moreover, we tested the end-to-end process discovery using the proposed technique and compared it to the traditional techniques. In traditional approaches, loading and mining time grow linearly as the event data grows. In contrast, our approach show constant times for updating (per event), while the retrieval and mining times are independent of the size of the underlying data.

The work presented in this paper is implemented in ProM. The plug-ins pave a way to access pre-computed DFR and MINERful relations stored in DB-XES. These relation values are then retrieved and processed by Inductive Miner and MINERful algorithms.

For future work, we plan to implement also the event removal and intermediate structures which are robust to filtering. The intermediate structures will be kept live under both insertion and deletion of events where possible. Furthermore, we aim to further improve the performance through query optimization and indexing.

References

1. A. Azzini and P. Ceravolo. Consistent Process Mining over Big Data Triple Stores. In *2013 IEEE International Congress on Big Data*, pages 54–61, June 2013.

2. 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.
3. C. Di Ciccio, F.M. Maggi, and J. Mendling. Efficient Discovery of Target-Branched Declare Constraints. *Information Systems*, 56:258 – 283, 2016.
4. C. Di Ciccio, F.M. Maggi, and J. Mendling. Efficient Discovery of Target-Branched Declare Constraints. *Information Systems*, 56:258 – 283, 2016.
5. C. Di Ciccio and M. Mecella. On the Discovery of Declarative Control Flows for Artful Processes. *ACM Trans. Manage. Inf. Syst.*, 5(4):24:1–24:37, January 2015.
6. C. Di Ciccio and M. Mecella. *Mining Constraints for Artful Processes*, pages 11–23. Springer Berlin Heidelberg, 2012.
7. C.W. Günther. XES Standard Definition. www.xes-standard.org, 2014.
8. S. Hernández, S.J. van Zelst, J. Ezpeleta, and W.M. P. van der Aalst. Handling Big(ger) Logs: Connecting ProM 6 to Apache Hadoop. In *BPM Demo Session 2015*, pages 80–84, 2015.
9. M. Jans, M. Alles, and M.A. Vasarhelyi. Process Mining of Event Logs in Internal Auditing: A Case Study. In *ISAIS*, 2012.
10. M.J. Jans, M. Alles, and M.A. Vasarhelyi. Process Mining of Event Logs in Auditing: Opportunities and Challenges. *Available at SSRN 2488737*, 2010.
11. 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.
12. 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.
13. 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.
14. F. Mannhardt. XESLite Managing Large XES Event Logs in ProM. *BPM Center Report BPM-16-04*, 2016.
15. R.S. Mans, H. Schonenberg, M. Song, W.M.P van der Aalst, and P.J.M Bakker. Application of Process Mining in Healthcare - A Case Study in a Dutch Hospital. In *BIOSTEC*, pages 425–438, 2008.
16. Z. Paszkiewicz. Process Mining Techniques in Conformance Testing of Inventory Processes: An Industrial Application. In *BIS Workshop*, pages 302–313, 2013.
17. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. chapter Linking Data to Ontologies, pages 133–173. Springer-Verlag, Berlin, Heidelberg, 2008.
18. M. Puchovsky, C. Di Ciccio, and J. Mendling. A Case Study on the Business Benefits of Automated Process Discovery. In *SIMPDA*, pages 35–49, 2016.
19. H. Reguieg, B. Benatallah, H.R.M. Nezhad, and F. Toumani. Event Correlation Analytics: Scaling Process Mining Using Mapreduce-Aware Event Correlation Discovery Techniques. *IEEE Trans. Services Computing*, 8(6):847–860, 2015.
20. A. Rozinat, I.S.M. de Jong, C.W. Günther, and W.M.P. van der Aalst. Process Mining Applied to the Test Process of Wafer Scanners in ASML. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 39(4):474–479, 2009.
21. 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.

22. V. Sharma and M. Dave. SQL and NoSQL Databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):20–27, August 2012.
23. W.M.P. van der Aalst. Distributed Process Discovery and Conformance Checking. In *FASE 2012*, pages 1–25, 2012.
24. W.M.P. van der Aalst. Decomposing Petri Nets for Process Mining: A Generic Approach. *Distributed and Parallel Databases*, 31(4):471–507, 2013.
25. W.M.P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.
26. W.M.P. van der Aalst and E. Damiani. Processes Meet Big Data: Connecting Data Science with Process Science. *IEEE Transactions on Services Computing*, 8(6):810–819, Nov 2015.
27. W.M.P. van der Aalst, Hajo A. Reijers, and Minseok Song. Discovering Social Networks from Event Logs. *Computer Supported Cooperative Work (CSCW)*, 14(6):549–593, 2005.
28. S. van der Spoel, M. van Keulen, and C. Amrit. Process Prediction in Noisy Data Sets: A Case Study in a Dutch Hospital. In *SIMPDA*, pages 60–83, 2012.
29. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. *Process Discovery Using Integer Linear Programming*, pages 368–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
30. B.F. van Dongen. BPI Challenge 2017, 2017.
31. B.F. van Dongen and S. Shabani. Relational XES: Data Management for Process Mining. In *CAiSE 2015*, pages 169–176, 2015.
32. 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.
33. 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.
34. T. Vogelgesang and H-Jürgen Appelrath. A Relational Data Warehouse for Multidimensional Process Mining.
35. 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*, pages 590–595, 2014.