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 a state-of-the-art process discovery technique. Secondly, we show how to move the computation of intermediate data structures, such as the directly follows relation, 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 [15]. 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 mining 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.

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,

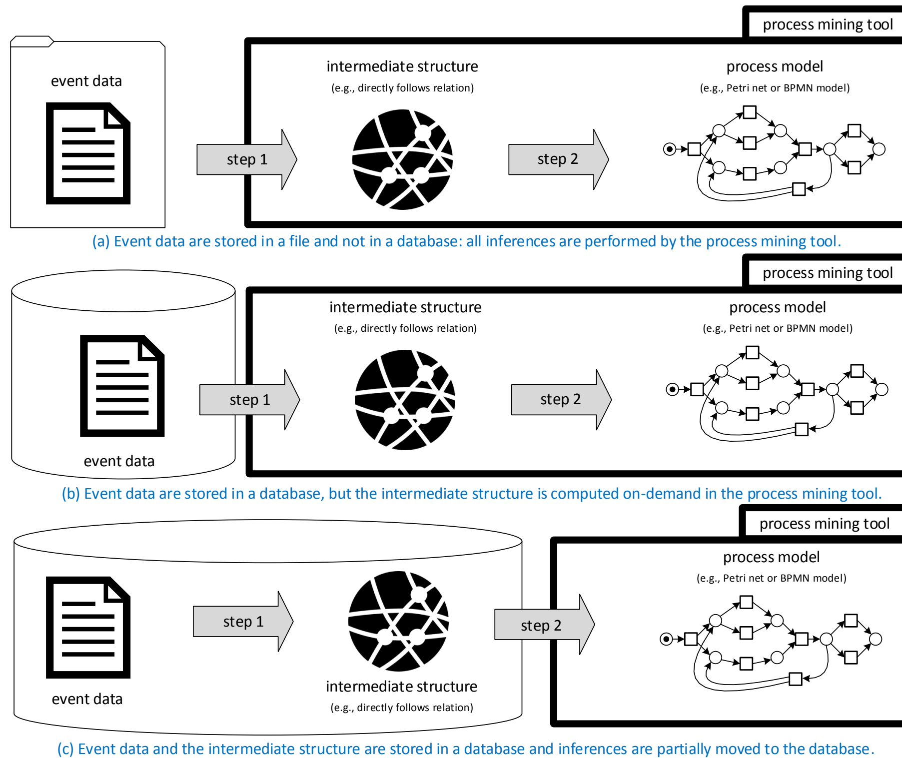

Fig. 1. Three different settings in process discovery

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 immediate structure in memory. Furthermore, the time needed for the analysis includes the time needed to convert the log to the intermediate structure.

To increase the scalability, relational databases have been proposed for storing event data [17], as depicted in Figure 1(b), i.e. the event log file is replaced by a database. In [17] 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 [6]. 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. In this paper we present the general idea and a concrete instantiation using the intermediate structure of a state-of-the-art process discovery technique. 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 structure. In Section 5 we show how a well-known intermediate structure can be computed inside the database. Then, in Section 6, we present experiments using the Inductive Miner. These show significant performance gains. Finally, we conclude and discuss the future work in Section 7.

2 Related Work

One of the first tools to extract event data from a database was XESame [20]. 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 [3] 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 [9]. 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 [17], 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

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

Lite implementations. However, its application to a real process mining algorithm was not investigated and the time-performance analysis was not included.

In [21], 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 procides a significantly better perfromance 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 [11] 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.

Apart from using databases, some other techniques for handling big data in process mining have been proposed [2, 10, 12], two of them are decomposing event logs [1] and streaming process mining [7, 18]. 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 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 [13]. 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 [6]. 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 [19], 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

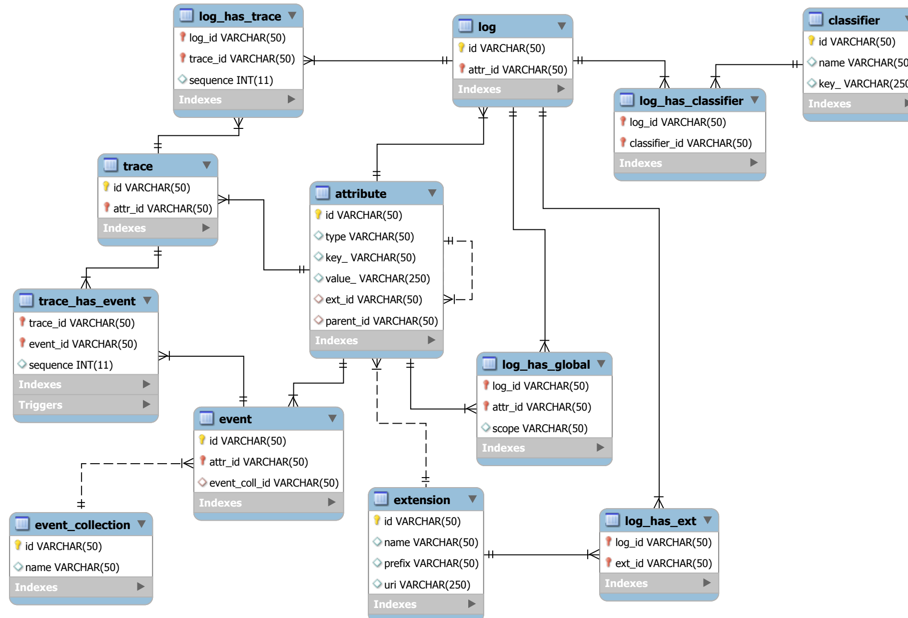

Fig. 2. DB-XES basic schema

which can be aggregated, but have difficulties supporting multiple perspectives at the same time. Besides, relational databases are more mature than NoSQL databases with respect to database features, such as 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 [6]. 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 a highly used intermediate structure we implemented in DB-XES as an example.

There are many existing intermediate structures in process mining, such as the eventually follows relation, no co-occurrence relation [4,5], handover of work relation [14], and prefix-closed languages in region theory [16]. 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 [15]. 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 [4, 5, 14], but some intermediate structures are bigger than the log [16]. 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

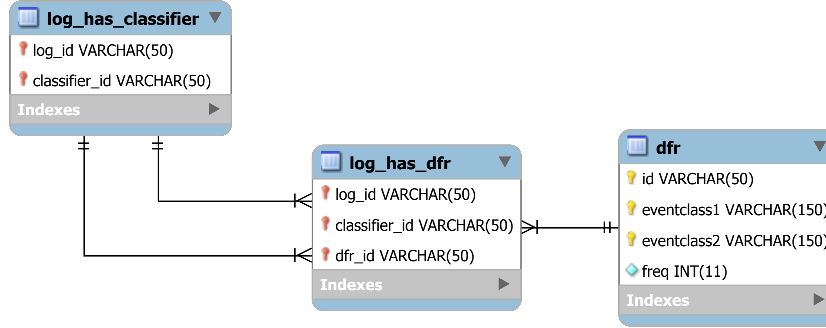

Fig. 3. DFR in DB-XES schema

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.

While many intermediate structures can be identified when studying process mining techniques, we currently focus on the *Directly Follows Relation (DFR)*. DFR is used in many process mining algorithms, including the most widely used process discovery techniques, i.e. Inductive Miner [8]. In the following we discuss how DB-XES is extended by a DFR table.

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 [15], DFR is defined as follows.

Definition 1 (Event Log). *Let E be a set 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 E be a set of events and let A be a set of attribute names.*

– *For any event $e \in E$ and name $a \in A$: $\#_a(e)$ is the value of attribute a for event e. $\#_a(e) = \perp$ if there is no value.*

– *Any subset* $C \subseteq \{a_1, a_2, ..., a_n\} \subseteq A$ *is a classifier, i.e., an ordered set of attributes. We define:* $\#_c(e) = (\#_{a_1}(e), \#_{a_2}(e), ..., \#_{a_n}(e))$.
– *In the context of an event log there is a default classifier* $DC \subseteq A$ *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* $1 \leq i < n$ *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 process discovery algorithm. However, DB-XES can be extended into other intermediate structures such as eventually follows relations and no co-occurrence relations.

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 to handle 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 between the database and process mining tools 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. This paper focuses on a particular intermediate structure, namely the DFR, 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.

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 exists 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 in 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                 /* Here is to get consecutive events */
10                WHERE t1.sequence = t2.sequence - 1
11             ) as temptable,
12             attribute as a1, attribute as a2,
13             event as event1, event as event2,
```

```
14                log_has_trace, log_has_classifier, classifier
15          WHERE <...>
16          GROUP BY log_id, classifier_id,
17                   event1.id, event2.id
18       ) as temptable2
19 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 10). The next step is obtaining the attribute values of these events. The attribute values are grouped based on the classifier in the log (line 16-17). 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 have the frequency of how often they appeared in the log (line 1, 19).

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 $(dfr_1, c, \bot, 1)$ 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 the next section.

6 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 Inductive Miner [8]. 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.

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. This experiment was executed on the machine with processor Intel(R) Core(TM) i7-4700MQ and 16GB of RAM.

6.1 Memory Use

In Figure 4(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.

In Figure 4(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 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.

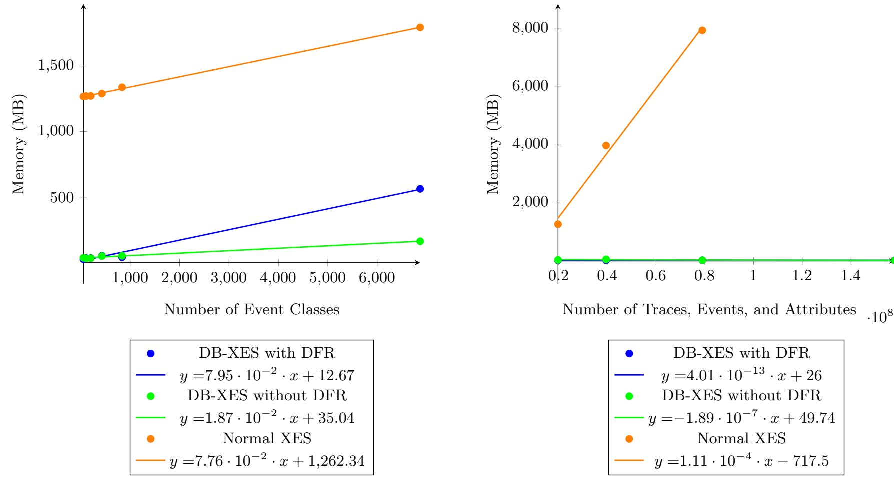

Fig. 4. From left to right: memory use of the Inductive Miner in: (a) logs with extended event classes and (b) logs with extended traces, events, and attributes

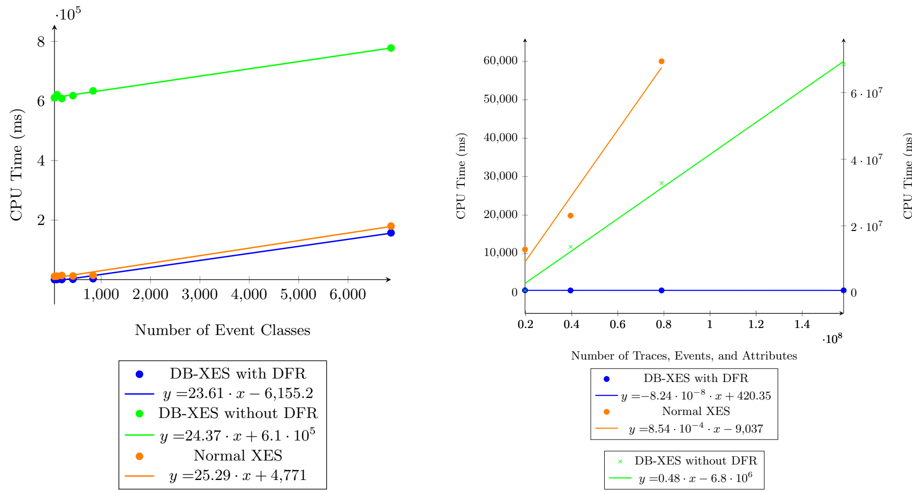

Fig. 5. From left to right: CPU time of the Inductive Miner in: (a) logs with extended event classes and (b) logs with extended traces, events, and attributes

6.2 CPU Time

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 5(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 5(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 computation of the intermediate structure to the database. This solution provides scalability in both memory and time.

We have implemented this solution as a ProM plug-in which connects DB-XES and Inductive Miner algorithm. We name the plug-in as *Database Inductive Miner* and it is distributed within the *DatabaseInductiveMiner* package (`https://svn.win.tue.nl/repos/prom/Packages/DatabaseInductiveMiner/Trunk/`).

7 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 directly follows relation can be stored in the same database and be kept up-to-date when inserting new events into the database.

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.

For the experiments we used the Inductive Miner, which is a state-of-the-art process discovery technique. However, the work trivially extends to other process discovery techniques, as long as we can identify an intermediate structure used by the technique which can be updated when inserting new events into the database.

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

For future work, we plan to implement also the event removal and intermediate structures which 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. W.M.P. van der Aalst. Decomposing Petri Nets for Process Mining: A Generic Approach. *Distributed and Parallel Databases*, 31(4):471–507, 2013.
2. 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.
3. Diego Calvanese, Marco Montali, Alifah Syamsiyah, and Wil MP van der Aalst. Ontology-driven extraction of event logs from relational databases. In *Business Process Intelligence 2015*. 2015.
4. Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. Efficient discovery of target-branched declare constraints. *Information Systems*, 56:258 – 283, 2016.
5. Claudio Di Ciccio and Massimo 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.W. Günther. XES Standard Definition. www.xes-standard.org, 2014.
7. Sergio Hernández, Sebastiaan J. van Zelst, Joaquín Ezpeleta, and Wil M. P. van der Aalst. Handling big(ger) logs: Connecting prom 6 to apache hadoop. In *BPM Demo Session 2015*.
8. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In *PETRI NETS 2013*.
9. Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Journal on data semantics x. chapter Linking Data to Ontologies, pages 133–173. Springer-Verlag, Berlin, Heidelberg, 2008.
10. Hicham Reguieg, Boualem Benatallah, Hamid R. Motahari Nezhad, and Farouk Toumani. Event correlation analytics: Scaling process mining using mapreduce-aware event correlation discovery techniques. *IEEE Trans. Services Computing*, 8(6):847–860, 2015.
11. Stefan Schönig, Andreas Rogge-Solti, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. *Efficient and Customisable Declarative Process Mining with SQL*, pages 290–305. Springer International Publishing, Cham, 2016.
12. W. v. d. 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.
13. Wil M. P. van der Aalst. Distributed process discovery and conformance checking. In *FASE 2012*.
14. Wil 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.
15. W.M.P. van der Aalst. *Process Mining: Data Science in Action*. 2011.
16. 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.
17. Boudewijn F. van Dongen and Shiva Shabani. Relational XES: data management for process mining. In *CAiSE 2015*.

18. Sebastiaan J. van Zelst, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Know what you stream: Generating event streams from CPN models in prom 6. In *BPM Demo Session 2015*.
19. Meenu Dave Vatika Sharma. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):20–27, August 2012.
20. 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 P. Soffer and E. Proper, editors, *Information Systems Evolution*, volume 72, pages 60–75, 2010.
21. Thomas Vogelgesang and H-Jürgen Appelrath. A relational data warehouse for multidimensional process mining.