XESLite - Managing Large XES Event Logs in ProM

Felix Mannhardt

Eindhoven University of Technology, Eindhoven, The Netherlands
f.mannhardt@tue.nl

Abstract. Process mining methods use data recorded by information systems to analyze the real execution of processes. This event data is stored in an event log, which is the main input to most process mining methods. The XES standard provides a uniform way to store event logs. OpenXES is the XES reference implementation, which is used widely by research tools. However, OpenXES is not scalable towards large event log. XESLite provides solutions to manage large event logs that are compatible with the OpenXES interfaces. Therefore, it can be used as drop-in replacement for existing algorithms. This report investigates the storage requirements of different types of event logs, describes XESLite, and contains a benchmark of XESLite and OpenXES based on real-life event logs.

Keywords: Event Logs, XES, Process Mining

1 Introduction

Organizations use process models to document, specify, and analyze their processes. The expected behavior of process is described in terms of its activities (i.e., units of work) and their ordering. Most modern-day processes are supported by information systems that record information about the execution of processes in databases. *Process mining* methods analyze processes by using this recorded data [2]. Most methods assume an *event log* as input. Event logs store the information about the executed activities in a structured manner. The recently published XES (eXtensible Event Stream) standard [1] provides a widely accepted view on what an event log entails. XES groups the *events* of an event log into *traces*[1] (i.e., sequences of events) and gives semantics to the *attributes* of an event. Figure 1 shows the complete meta model of XES. Moreover, the XES standard defines an XML-based serialization format to achieve interoperability between tools. The open-source reference implementation OpenXES[2] simplifies reading and manipulating the XML-based serialization format for event logs from Java programs. Therefore, OpenXES is widely used among research tools, e.g.,

[1] The XES standard also describes an ungrouped stream of events, i.e., all events are grouped into a single trace. For XESLite, we consider the offline setting.

[2] http://www.openxes.org

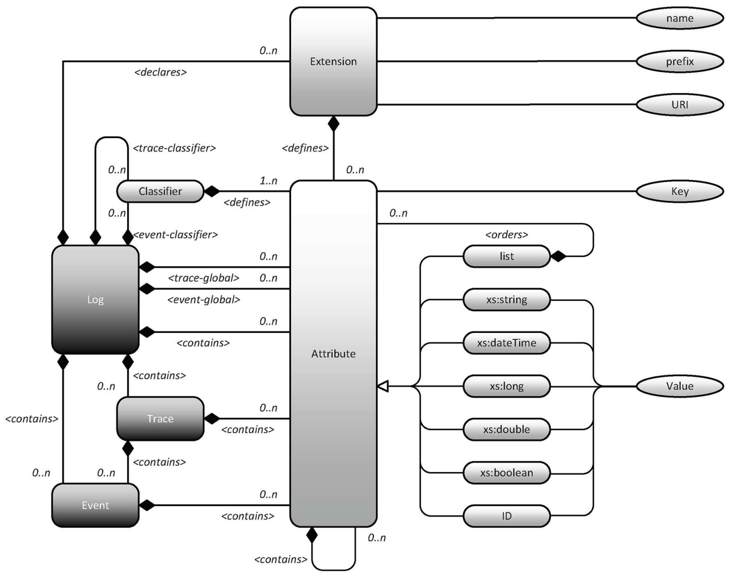

Fig. 1. Meta model of the XES standard as proposed in [1].

in the leading open-source process mining framework ProM [3] and the process model repository Apromore [4].

OpenXES stores the event log in main memory[3] using the standard classes provided by the Java collections framework. This simplifies the development and works well with small event logs typically used as examples for research purposes. However, when using OpenXES for large and complex real-life event logs (e.g., 10 million events with 3 attributes each), the available main memory on a typical workstation (e.g., 4 GB) is insufficient to load the event log. Generally, there are two directions that can be taken to solve this problem:

- store the event log more *efficiently in the main memory*, or
- store the event log *outside the main memory* and fetch parts of it on-demand.

Within the academic world work has been dedicated to the latter direction. For example, by implementing a XES interface to a database organized along a specific event log schema [6] or through a buffered implementation for event logs [5]. Moreover, there is some recent work on reducing the need for data transfer between the process mining tool and the database by offloading the computation of data structures to the database. For example in work [7] a directly-follows graph is built within the database and in work [8] declarative constraints are discovered using SQL directly on the database.

[3] In older versions of OpenXES there was a solution [5] to buffer full traces of the event log to the disk. However, this solution was removed in v2.0 due to a bug.

Commercial tools support very large event logs (e.g., Celonis[4] uses the SAP HANA in-memory architecture). However, for commercial tools the access pattern of the mining algorithm and the data structure can be heavily optimized towards each other. In contrast, OpenXES needs to be flexible, open, and support many different access patterns that are being used by academic research prototypes.

XESLite provides solutions to manage large event logs, which are compatible with the OpenXES interfaces. XESLite can be used as drop-in replacement of OpenXES. Thus, it can be used to apply existing research software to large event logs. XESLite is not suited for 'Big Data' event logs, with storage requirements in the ranges of terabytes. XESLite focuses on typical real-life event logs found in a research environment with up to 100 million events[5]. For example, in the IEEE Taskforce on Process Mining repository of event logs[6], there are currently 12 publicly available events logs for research purposes (e.g., [9,10,11]). The largest event log contains about 7 million events with at most 20 attributes per event [10]. We present three XESLite variants, each optimized for a certain type of event log:

– XESLite - Automaton (XL-AT),
– XESLite - In-Memory (XL-IM), and
– XESLite - Database (XL-DB).

The remainder of this report is structured as follows. First, we state the problem in a more systematic manner and introduce the requirements for storing event log (Section 2). Then, we introduce each of the three XESLite variants in more detail (Section 3). Finally, we test XESLite on several real-life event logs (Section 4) and conclude the paper (Section 5).

2 Event Logs

2.1 What is an Event Log?

An event log stores information about activities that were recorded by information systems while supporting the execution of a process. Each execution of a *process instance* results in a sequence of events. Events stored in the event log relate to activities of the process. Furthermore, events may store data produced by the activities. This data is stored in attributes that are associated with an event. We define an event log as set of unique events that are assigned attributes [2,1]. For the presentation of XESLite we choose a simplified view of event logs. In contrast to the definition in [2], we do not consider case attributes and activity classifiers. Moreover, we do not consider the role of extensions, classifiers, and nested attributes[7].

[4] `http://www.celonis.com`

[5] The maximum number of events supported depends on the number of attributes (i.e., the event payload) and the structure of the data.

[6] `http://data.4tu.nl/repository/collection:event_logs_real`

[7] Still, the XESLite variants XL-IM and XL-DB support storing nested attributes.

Table 1. An excerpt of an event log $(E, \Sigma, \#, \mathcal{E})$. Each event is identified by a unique identifier **id** and records attributes **case**, **activity**, **time**, **lifecycle**, **amount** and **granted**. We use symbol $\perp$ to denote an unrecorded attribute. Traces $\mathcal{E}$ can be obtained by grouping events according to their **case** attribute.

id	**case**	**activity**	**time**	**lifecyle**	**amount**	**granted**
e_1	1	Register	2016-01-01T10:00:00	start	$\perp$	$\perp$
e_2	1	Register	2016-01-01T10:02:00	complete	$\perp$	$\perp$
e_3	1	Apply for Credit	2016-01-01T10:12:00	start	$\perp$	$\perp$
e_4	1	Apply for Credit	2016-01-01T10:12:05	complete	10000	$\perp$
e_5	1	Check	2016-01-02T09:00:00	complete	$\perp$	`true`
e_6	1	Send Contract	2016-01-03T16:00:00	start	$\perp$	$\perp$
e_7	1	Send Contract	2016-01-03T16:00:01	complete	$\perp$	$\perp$
e_8	2	Register	2016-01-02T10:00:00	start	$\perp$	$\perp$
e_9	2	Register	2016-01-02T10:02:00	complete	$\perp$	$\perp$
e_{10}	2	Apply for Credit	2016-01-02T12:30:00	start	$\perp$	$\perp$
e_{11}	2	Apply for Credit	2016-01-02T12:31:00	complete	1000	$\perp$
e_{12}	2	Check	2016-01-02T16:00:00	complete	$\perp$	`false`
e_{13}	2	Inform Rejection	2016-01-03T08:00:00	start	$\perp$	$\perp$
e_{14}	2	Inform Rejection	2016-01-03T08:00:01	complete	$\perp$	$\perp$
e_{15}	3	Register	2016-01-05T10:00:00	start	$\perp$	$\perp$
e_{16}	3	Register	2016-01-05T10:02:00	complete	$\perp$	$\perp$
e_{17}	3	Apply for Credit	2016-01-05T10:12:00	start	$\perp$	$\perp$
e_{18}	3	Apply for Credit	2016-01-05T10:12:05	complete	5000	$\perp$
e_{19}	3	Check	2016-01-06T09:00:00	complete	$\perp$	`true`
e_{20}	3	Send Contract	2016-01-07T16:00:00	start	$\perp$	$\perp$
e_{21}	3	Send Contract	2016-01-07T16:00:01	complete	$\perp$	$\perp$
e_{22}	4	Register	2016-01-12T10:00:00	start	$\perp$	$\perp$
e_{23}	4	Register	2016-01-12T10:02:00	complete	$\perp$	$\perp$
e_{24}	4	Un-register	2016-01-13T12:15:00	start	$\perp$	$\perp$
e_{25}	4	Un-register	2016-01-13T12:16:00	complete	$\perp$	$\perp$
e_{26}	5	Register	2016-01-14T08:21:00	start	$\perp$	$\perp$

Definition 1 (Event Log). *Given universes of attributes A and values U, we define an event log as $(E, \Sigma, \#, \mathcal{E})$ with:*

- *E is a finite set of unique event identifiers;*
- *$\Sigma \subseteq U$ is a finite set of activity names;*
- *$\# : E \rightarrow (A \not\rightarrow U)$ retrieves the attribute values assigned to an event;*
- *$\mathcal{E} \subseteq E^*$ is the finite set of traces over E. A trace $\sigma \in \mathcal{E}$ records the sequence of events for one process instance. Each event only once, i.e., the same event may not appear twice in a trace or in two different traces.*

Given an event $e \in E$ in the event log $(E, \Sigma, \#, \mathcal{E})$, we write $\#_a(e) \in U$ to obtain the value $u \in U$ recorded for attribute $a \in A$. One mandatory attribute is recorded by each event: $\#_{activity}(e) \in \Sigma$, the *name of the activity* that caused the event.

Example 1. Table 1 shows an excerpt of an example event log $(E, \Sigma, \#, \mathcal{E})$. Each row represents a unique event $e \in E$ together with the produced data (i.e., attributes). For example event e_1 recorded the execution of the activity Register (i.e., $\#_{activity}(e_1)$ = Register). Moreover, it recorded additional attributes $\#_{case}(e_1) = 1$, $\#_{time}(e_1)$ = 2016-01-01T10:00:00 and $\#_{lifecycle}(e_1)$ = start. For the remaining two attributes, amount and granted, no value ($\perp$) has been recorded. The attribute *case* can be used together with the attribute *time* to group events into the set of traces $\mathcal{E}$. For example, we obtain the trace $\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7 \rangle$ for the case identifier 1. All events in a trace are ordered by the attribute *time*.

An event log is a (potentially sparse) table of values. Some columns (i.e. attributes) of the table (i.e, case, event and time) are assigned special semantics. Those attributes allow to view the event log as set of ordered traces. The other attributes are not strictly necessary for many basic process discovery algorithms. Still, additional attributes provide important contextual information. As described by van der Aalst [2], we can obtain a *simplified event log* based on the minimal set of attributes: case, event, and time. Please note that the attribute *time* is not strictly needed if events are ordered according to the execution time of activities. Given a set X, $\mathbb{B}(X)$ denotes the set of all multi-sets over set X.

Definition 2 (Simple Event Log). *Given a universe of activity names Σ. A simple trace σ is a sequence of activity executions, i.e., $\sigma \in \Sigma^*$. A simple event log L is a finite multi-set of traces over Σ, i.e., $L \in \mathbb{B}(\Sigma^*)$.*

A large number of process discovery methods (e.g., Inductive Miner [12], Heuristics Miner [13], and Fuzzy Miner [14]) require only a simple event log as input. However, information on contextual factors such as time, resources, life-cycle transitions and data attributes is lost. Therefore, a simple event log is not suited more advanced process mining methods (e.g., performance measurements and decision mining). Still, process models can be discovered from simple event logs.

Example 2. We can transform the event log shown in Table 1 into a simple event log. First, we remove all events with the life-cycle transition start, i.e., we only keep events for the completion of an activity. We remove every attribute except the activity name from all events. Moreover, we remove the event identifier, i.e., the ability to uniquely identify traces and events. We abbreviate each activity name with its first character (e.g., Register is abbreviated with R). Then, the event log shown in Table 1 is transformed to the simple event log $L = [\langle \mathrm{R}, \mathrm{A}, \mathrm{C}, \mathrm{S} \rangle^2, \langle \mathrm{R}, \mathrm{A}, \mathrm{C}, \mathrm{I} \rangle, \langle \mathrm{R}, \mathrm{U} \rangle, \langle \mathrm{R} \rangle]$.

2.2 How Much Memory is Really Required?

We have defined the kind of information that needs to be stored in event logs that we want to store. Now, we estimate the minimal memory requirement for storing a normal event log $(E, \Sigma, \#, \mathcal{E})$ according to Definition 1. Moreover, we determine how much overhead is introduced by OpenXES.

We use a simple estimation of the minimal memory requirements for an event log with n events (i.e., $n = |E|$) and m attributes (i.e., $m = |dom(\#)|$). We *ignore*

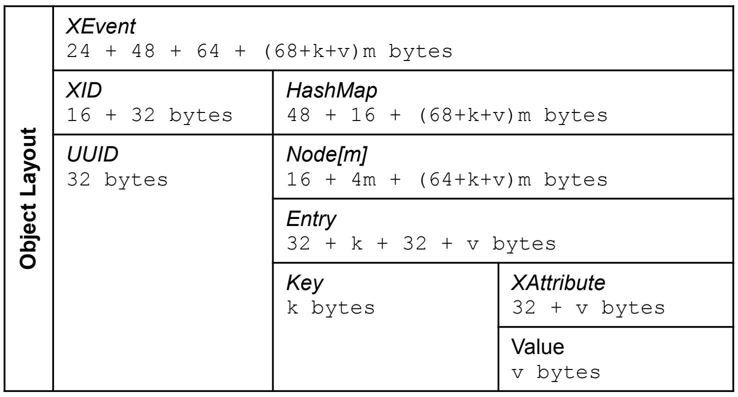

Fig. 2. Object layout and memory consumption of an event in the OpenXES reference implementation. We assume that the event contains x attributes and that it takes k bytes to store the key and v bytes to store the value of an attributes. Then, each event consumes at least $136 + (68 + k + v)m$ bytes of memory. The estimation is based on the typical Java run-time environment on a 64 bit architecture.

any additional overhead that is used to store the trace structure and we *assume that all attributes are recorded for every event*, i.e., we see the event log as a dense table with dimensions $n \times m$. Assuming that it takes v bytes to store the value of a single attributes, then, such a table-based event log would minimally require $n \cdot m \cdot v$ bytes of memory. This calculation assumes *no compression* based on re-occurring attribute values and ignores the memory used to store the column headers. For the practical use of any event log additional data structures, which support querying and modifying the trace structure and individual attributes, need to be stored. However, this simple calculation provides us with a realistic underestimation on the memory overhead of the current reference implementation. Moreover, it can be used to estimate the sizes of event logs that can be stored in the main memory of a typical workstation.

We estimated the memory requirement of OpenXES by inspecting its implementation[8] and using a profiler to determine the memory size of a single event with m attributes. Figure 2 shows the memory layout of the *XEvent* implementation in OpenXES. Each event contains a unique identifier that is stored in the *XID* class and a map of attributes, which is implemented as standard Java *HashMap*. Each attribute is wrapped in both an *XAttribute* object and an *Entry* object. In total, the OpenXES implementation uses $136 + (68 + k + v)m$ for each event.[9] This constitutes an overhead of 136 bytes of each event and an overhead of 68 bytes for each attribute. Moreover, the key (i.e., name) of the attribute is stored separately along each value.

[8] OpenXES v2.15 (`http://www.xes-standard.org/openxes/download`)

[9] We ignore the additional memory overhead of the empty slots in the map in this estimation, i.e., we consider a full map with a load factor of 1.0.

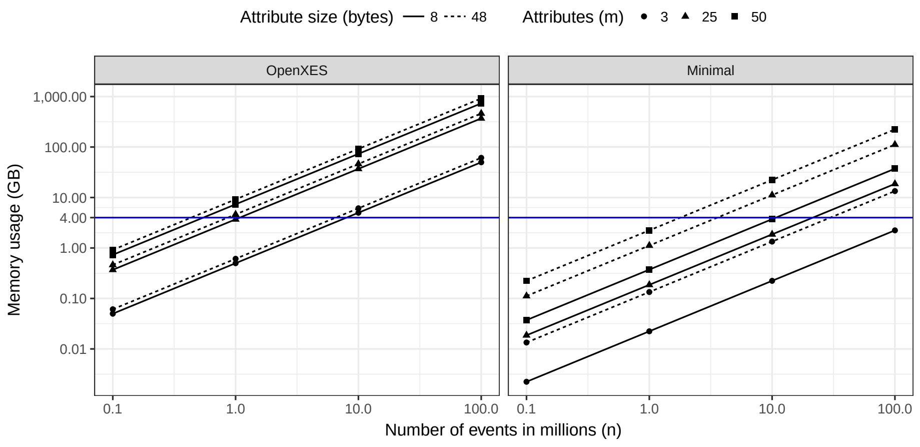

Fig. 3. Comparison between the OpenXES memory requirements and the minimal memory requirements for an event log with three numbers of attributes (3, 25, and 50) and two attribute sizes (8 byte and 48 byte). The values underestimate the actual memory usage since the memory required for storing the trace structure is omitted.

Figure 3 shows the estimated memory requirements of the OpenXES reference implementation compared with the minimal memory requirement based on storing a dense table of $n \times m$ attributes for two attributes sizes (8 bytes and 48 bytes) and three values for the number of attributes m. The memory requirements for **small event logs** of up to 1,000,000 events, **medium-sized event logs** of up to 10,000,000 events, and **large event logs** of up to 100,000,000 events are shown. As motivated in Sect. 1, we do not consider larger event logs as those are not targeted by XESLite. The blue horizontal line is placed on 4 GB, a typical amount of available memory in current work stations. The overhead imposed by the OpenXES implementation is visible in Figure 3. For example, there is little difference between an event log storing attribute values of 8 bytes (e.g., a time-stamp or a discrete number) compared to an event log storing attribute values of 48 bytes (e.g., a literal values). Most of the memory consumption is based on the overhead added by the Java objects and not based on storing the actual data. In fact, when considering the minimal memory requirements it would be possible to store an event log containing 100,000,000 events with 3 attributes with 8 bytes values in the main memory of a typical work station. Please note that this calculation does not consider compression strategies that are possible when the data stored in the attributes is not random. Certainly, in typical event logs the data is far from being random.

We observe that the OpenXES reference implementation introduces large amounts of memory overhead per event. Moreover, we observe that the actual data stored in large event logs can fit reasonably within the memory limits of a typical work station. However, when naïvely viewing the event log as one large

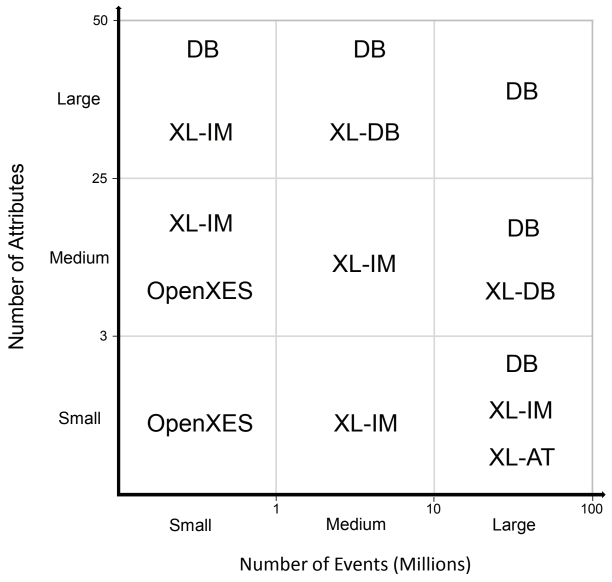

Fig. 4. Suitability of XES implementations for event logs of different sizes based on two dimensions: the number of attributes per event and the number of events. Each quadrant is assigned a preferred implementation, i.e., the fastest XES implementation for the event log on a machine with 4 GB available memory.

table of attribute, the query, manipulation, and other extended functionality provided by OpenXES would be missing. The goal of XESLite is to keep the memory consumption of such large event logs close to the minimal memory requirement while retaining most of the functionality as defined by the OpenXES interface.

2.3 Which XES Implementation to Choose?

Having investigated the theoretical and the actual memory requirements to store event logs, we classify the suitability of XES implementations for event logs of different sizes along the two dimensions: number of event and number of attributes. We consider the three XESLite variants (XL-AT, XL-IM, and XL-DB) that are presented in the next Section 3 as well as the reference implementation OpenXES. Moreover, we also consider the option, in which the event log is stored in an external database, and algorithms are tailored towards accessing or manipulating the data directly from this database (e.g., as proposed by [6,8,7]). Figure 4 provides an overview on the suitability of XES implementations for different groups of event logs: small, medium, and large in the number of events combined with small, medium, and large in the number of attributes. We estimate the memory requirements for event logs in each of the nine groups. We assume that the attribute payload is small (i.e., 8 bytes). We deem a XES implementation to be suitable if it can handle the event logs within the group on a typical work station with 4 GB of memory.

OpenXES For event logs with a small number of events and small or medium number of attributes (i.e., an event log with less than 1,000,000 events and less than 25 attributes), the OpenXES implementation can be used. If attributes contain bigger payload (e.g., 48 bytes literals), then, it may not be possible to use the OpenXES implementation anymore. For event logs that are larger in the size of events or attributes, the OpenXES implementation is clearly unsuited.

DB On the other hand of the spectrum, there are very large event logs with many events and many attributes. Those event logs cannot be stored in the main memory of a typical work station. These event logs should be stored in an external database. Techniques such as [6] or algorithm specific solutions such as [8,7] could be employed.

XL-AT This specialized automaton-based XESLite variant supports only simple event logs according to Definition 2. It is suited for event logs with a very large number of events since it encodes common prefixes and suffixes of traces in an efficient manner.

XL-IM The in-memory XESLite variant stores the attributes of events in a compressed tables that are organized in a column-format. Attribute with the same key are stored together. XL-IM is suited for a wide variety of event logs ranging from event logs with a large number of events and small number of attribute to event logs with a large number of attributes and a small number of events.

XL-DB This XESLite variant is based on an embedded database. The trace structure (i.e. E and $\mathcal{E}$) is stored in-memory whereas the attributes (i.e., $\#$) are stored in the embedded database. The implementation is based on MapDB[10] a hybrid between an embedded database engine and a disk-backed replacement for the Java collections framework. This variant is useful for storing medium to large event logs with many attributes. Because the trace structure is stored in-memory, XL-DB can be very efficient when only a small subset of attributes is accessed frequently.

Please note that this categorization is only a rough estimation of the suitability. Depending on the exact shape of the data stored in the event log, the suitability can differ. In the following Section 3, we describe the inner workings of the three XESLite variants in more detail.

3 XESLite

XESLite provides three implementations of the XES Java interfaces that are defined by OpenXES. XESLite is available as package of the open-source process mining framework ProM[11] and as a stand-alone implementation[12].

[10] `http://www.mapdb.org`

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

[12] `https://github.com/fmannhardt/xeslite`

3.1 General Ideas

There are some general ideas that are used across all XESLite variants: using flyweight literals, using sequential identifiers for events, and using a compressed encoding for traces.

Flyweight literals In many event logs, literals are used as values in attributes. Typically, only a small set of literal values is used and repeated across the event log. For example, in Table 1 only two distinct literals are used for the lifecycle attribute: “start” and “complete”. When stored naïvely, each attribute values referring to either the value “start” or the value “complete” is stored independently. For the event log in Table 1, the literal “start” is stored 9 times and the literal “complete” 12 times. XESLite uses the flyweight strategy [15] to avoid storing duplicates of identical literals. Literals are stored in a table and references to the literal in the look-up table are stored instead of duplicate values. For typical event logs, this leads to a considerable **memory reduction that depends on the amount of duplicate literals**. However, the universe of values for most attributes is typically much smaller than the number of events. The drawback of this strategy is overhead when inserting new literals into the look-up table.

Sequential identifiers In the OpenXES reference implementation every event is assigned a unique identifier *XID*. This matches Definition 1 of an event log as a set of unique events. However, the identifier is implemented as *XID* object that contains a unique random *UUID* object. As shown in Figure 2, this implementation choice leads to an overhead of 48 bytes per identifier. Therefore, we replaced the *XID* with a sequential counter that is implemented with a primitive *long* (i.e., 8 bytes). This restricts the size of an event log in XESLite to 2^{64} events, which is not a limitation in practice. Using sequential identifiers results in **memory savings of 40 bytes per event**.

Compressed traces We can describe traces of events as a sequence of identifiers. For example, the trace $\sigma = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7 \rangle$ can be described by its projection on the indicies $\sigma_{ids} = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$. Thus, the traces of an event log $\mathcal{E}$ are a set of sequences of indicies. For example, for log introduced in Table 1 we obtain $\mathcal{E}_{ids} = \{\langle 1, 2, 3, 4, 5, 6, 7 \rangle, \langle 8, 9, 10, 11, 12, 13, 14 \rangle, \langle 15, 16, 17, 18, 19, 20, 21 \rangle, \langle 22, 23, 24, 25 \rangle, \langle 26 \rangle\}$. Here, we abstract from the attributes of events that need to be stored elsewhere. Typically, events are ordered by traces in the XES serialization format. Events within a trace can be assigned subsequent numbers as identifiers. For such sequences, storing only the start index and the length of the sequence (i.e. 16 bytes per trace) would be sufficient to describe the trace structure of the event log. However, in practice we have to support deletions and insertions, thus, we cannot guarantee such perfect sequences. Therefore, we first apply *delta encoding* on the sequences (e.g. the delta encoding of σ_{ids} is $\langle 1, 1, 1, 1, 1, 1, 1 \rangle$) and compress the result with LZ4[13]. There is work on more efficient representations of such sequences of integers (e.g., [16]), however, the size

[13] `http://www.lz4.org/`

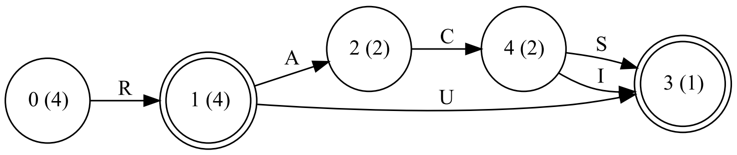

Fig. 5. The minimal DAFA generated from the simple event log L. The number of suffixes that are reachable from a state is placed in parenthesis after the state identifier.

of the trace structure is not an issue for the event logs that we encountered in practice. The **exact memory savings of using compressed traces depend on the event log at hand**.

3.2 XESLite - Automaton

The variant *XESLite - Automat* (XL-AT) is suited for very large event logs with events, for which only the activity name is required. In fact, many process discovery algorithms (e.g, Inductive Miner [12], Heuristics Miner [13], and Fuzzy Miner [14]) only require a simple event log according to Definition 2. Therefore, there are scenarios, in which it can be sufficient to discover the control-flow of a process model on a very large simple event log. Only afterwards, the other attributes such as timestamps and resources are used for conformance checking and decision mining.

A simple event log is a multi-set of traces, where each trace is a sequence of activity names. For example, multi-set $L = [\langle \mathrm{R}, \mathrm{A}, \mathrm{C}, \mathrm{S}\rangle^2, \langle \mathrm{R}, \mathrm{A}, \mathrm{C}, \mathrm{I}\rangle, \langle \mathrm{R}, \mathrm{U}\rangle, \langle \mathrm{R}\rangle]$ is the simplified version of the event log in Table 1. We can also see such a simple event log L as the multi-set of words over an alphabet Σ, i.e., trace $\sigma \in \Sigma^*$. Then, the set of words $W \subseteq \Sigma^*$ that is contained in L, i.e., $W = \{\sigma \in L\}$, can be described by a finite regular language over Σ. In fact, we required L to be finite and every finite language is regular [17].

Words of a finite regular language (i.e., set W) can be describe with an Deterministic Acyclic Finite Automaton (DAFA). DAFAs can be minimized [17] to a unique minimal DAFA to reduce the storage requirements. Moreover, it is possible to efficiently build the minimal DAFA incrementally based on a set of lexicographically ordered words [18]. In XESLite, we make use of this method.

Figure 5 shows the minimal DAFA accepting only the language W. However, such a DAFA does not store multiplicities of words, i.e. it cannot directly be used to store L. We need to store the multiplicities of words (i.e. traces) elsewhere, e.g., in a look-up table such as shown in Table 2. To link the look-up table with the words in the DAFA, each word in the language of the DAFA needs to be assigned a unique index. In fact, a DAFA can be extended with a simple numbering scheme to provide such a perfect hash function [19,20]. Work in the natural language processing domain [19,20], describes a method that assigns unique consecutive numbers $1 \ldots n$ to the words accepted by a DAFA. The basic idea is to assign the numbers to words in their lexicographical order, i.e., a word

Table 2. Look-up table for the multiplicities of words accepted by the DAFA. The index is based on the lexicographical order of the words. We added the column trace for illustration only, the traces do not need to be stored.

index	**multiplicity**	**trace**
1	1	$\langle R \rangle$
2	1	$\langle R, A, C, I \rangle$
3	2	$\langle R, A, C, S \rangle$
4	1	$\langle R, U \rangle$

is assigned the number of its predecessors in the lexicographical ordering of the language. These numbers can be computed on the DAFA by storing a number on each state of the automaton.

We assign state the number of words that would be accepted by the automaton starting from this state [19]. In other words, we assign the number of words in the right language and we add one to the number if the state is a final state. Figure 5 shows such a numbered automaton for the event log L. For example, in Figure 5 we assign the number 4 to state 0 since there are three words in its right language and it is a final state. State 2 is assigned the number 2 since there are two words its right language. Based on this automaton, Lucchhesi et al. [19] present a simple algorithm to determine the unique consecutive indicies that are shown in Table 2.

Given a word $w \in W$, we use the numbered states to determine how many words $w' \in W$ precede word w in the language W, i.e., $|\{w' \in W \mid w' < w\}|$. Please note that this method requires a total order on the alphabet on the language. It is easy to define such a total order for activities of an event log, e.g., we can use the alphabetical order of the activity names. We sketch the basic idea using the word $\langle R, U \rangle$ as an example.

Example 3. For word $\langle R, U \rangle$ in our example log there are three preceding words: $\{\langle R \rangle, \langle R, A, C, I \rangle, \langle R, A, C, S \rangle\}$. We can calculate index 3 for word $\langle R, U \rangle$ by parsing the word with the automaton in Figure 5. For each traversed state, we add the numbers assigned to those neighboring states that would be parsed for symbols in the alphabet that precede the next symbol in our word. In fact, we know that these states eventually lead to words that are lexicographically smaller the currently parsed word. Moreover, we add one to the count when traversing a final state. Final states along our path through the automaton correspond to words that are shorter and, thus, lexicographically smaller than the currently parsed word. Let $i \in \mathbb{N}_0$ be the counter for the index. We start in state 0 and parse symbol R. Nothing is added to i since the only available transition is R. Being in state 1, we assign $i = 1$ as state 1 is a final state. In fact, a preceding word $\langle R \rangle$ would be accepted here. Moreover, from state 1 we can reach state 2 by firing transition A. Since A precede symbol U, we add the number of state 2 to the counter i, i.e., $i = 1 + 2 = 3$. There are no other transitions preceding the next symbol U, therefore, we execute transition U and reach the final state

3. Again, we add one to the counter and obtain the index for word $\langle R, U\rangle$, i.e., $i = 3+1 = 4$. Similarly, we can find words in the language given their index. We refer to [19] for the details.

The XL-AT variant efficiently encode a simple event log with a DAFA and a corresponding multiplicities look-up table. We use the open-source library Dicomaton[14], for the implementation of the just describes perfect hashing scheme. Since the traces of events logs often share prefixes and suffixes, XL-AT can efficiently encode very large logs. Still, there are some limitations to this approach. The implementation of XL-AT does not yet support any modification of the traces and requires added traces to be lexicographically sorted. Moreover, the compression of common prefixes and suffixes in a DAFA does not work for event logs that record traces of systems with a large number of activities that are executed in parallel. Traces from those systems are unlikely to share prefixes and suffixes with other traces.

3.3 XESLite - In-Memory

The variant *XESLite - In Memory* (XL-IM) is suited for medium to large event logs. XL-IM supports event logs according to Definition 1, i.e., events are uniquely identified and can be associated with attributes. Since all data is stored in main memory the size of the event log is limited by the available memory. However, XL-IM uses a memory efficient column-layout to minimize the overhead associated with storing attributes.

As motivated in Section 2.2, an event log $(E, \Sigma, \#, \mathcal{E})$ with n events and m attributes can be seen as a table with dimensions $n \times m$. XL-IM adopts this tabular view of an event log. Since a typical event log contains many more event than attributes (i.e., $n \gg m$), XL-IM represents attributes as the columns of the table and events as the rows. Events are addressed by their sequential identifier and attributes by a mapping from attribute names to a column index. By doing so, we can optimize the storage for each attribute in two ways:

1. we enable the compression of similar values and
2. we enable a quick look-up of values with by using fixed-width values without additional index overhead.

The ideas are in some way related to the idea of column-stores [21]. For XESLite, we only exploit the compression benefits of a storage layout organized by columns.

A requirement of XL-IM is that attributes are assigned values from consistent subsets of the universe of values, i.e., for each attribute $a \in A$, we can determine a subset $U_a \subseteq U$. In fact, the XES standard describes several types of attributes (e.g., discrete, boolean, and literal). For example, in Table 1, the attribute *amount* is assigned values from the universe of integers, the attribute *lifecycle* is assigned values of the universe of literals and the attribute *granted* is

[14] `https://github.com/danieldk/dictomaton`

Table 3. A block of size 2, which stores the integer values of the attribute *amount*. Bytes 0 until 7 are used to store two 4 bytes integer values, which are shown encoded in big-endian order. Byte 8 is used to store flags indicating whether the attribute is not set (i.e., $\perp$).

content	byte	value (byte)	value (integer)
e_3	0	00	0
	1	00	
	2	00	
	3	00	
e_4	4	00	10000
	5	00	
	6	27	
	7	10	
flags	8	0 | 1 | 0 | 0 | 0 | 0 | 0 | 0	

assigned values from the universe $\{\texttt{true}, \texttt{false}\}$. By restricting the universe of attribute a to a fixed sub-set U_a, we can optimize the storage format of values for attribute a.

We store the values of the attributes in blocks. Each block stores a fixed number of fixed-width values. Each block is an array of bytes (i.e., the Java type byte[]). We use multiple blocks of configurable size for each attribute such that the storage can be expanded concurrently. Blocks store a fixed number of values and a set of flags, which indicate the presence or absence of a value (i.e., the symbol $\perp$). Currently, the implementation of XL-IM supports the four widths: 1 bit, 4 bytes, 8 bytes, and 16 bytes. Variable-width literal values are stored by using 4 byte references, which point to rows in a look-up table of literals. A look-up table for literals is kept to apply the flyweight pattern anyway.

For example, Table 3 shows a small storage block for the attribute *amount*. The storage block contains the values of two events: e_3 and e_4. We assume that the discrete values of attribute *amount* can be encoded with 4 bytes. The whole block requires 9 bytes of memory: 8 bytes to store the two 4 byte values and 1 byte to store the additional flags. Only e_4 records a value for the attribute *amount*, i.e., $\#_{amount}(e_3) = \perp$ and $\#_{amount}(e_4) = 10000$. Therefore, the first four bytes are left at their default value 0 and the first bit of the flag byte 8 is set to 0. The next four bytes 4 to 7 encode the value 10000 and the second bit of the flag byte is set to 1. The remainder of the bits of the flag byte remain unused in this example. In practice, we use a multiple of eight as block size, thus, every bit of the flag bytes is used.

A second example for a storage block of XL-IM is given in Table 4. Here, boolean values of the attribute *granted* are stored. The block of size 8 uses two bits for each boolean value. The value bit in the first byte stores the actual boolean value and the flag bit in the second byte stores the information about whether the attribute is set (i.e., $\perp$).

Table 4. A block of size 8 that stores the boolean values of the attribute *granted*. Each bit of bytes 0 is used to store the eight boolean values. The second byte is used to store flags indicating whether the attribute is not set (i.e., $\perp$).

content	**byte**	**value** (bit)							
$e_5 \dots e_{12}$	0	1	0	0	0	0	0	0	0
flags	1	1	0	0	0	0	0	0	1

This storage layout is much more compact than the one used by OpenXES (cf. Figure 2) since it avoids all the overhead associated with using objects and maps for attributes. Moreover, storing the values in a contiguous block of memory allows to pack together several small-width values (e.g., booleans) in a single word (i.e., 8 byte on a 64 bit machine). Otherwise, each value would at least occupy a full word-size of memory. Finally, the storage layout of XL-IM allows us to apply compression techniques (e.g., LZ4) to the blocks. By storing similar values together in a block, data compression can be particularly effective. Moreover, it is possible to trade-off memory savings against the computation overhead by changing the block size. Currently, XL-IM does not implement a garbage collection mechanism, i.e., the cells occupied by deleted attribute values are not re-used. This is not a severe limitation since event logs are considered immutable by most plug-ins of ProM.

3.4 XESLite - MapDB

The variant *XESLite - MapDB* (XL-DB) is suited for large event logs. XL-IM supports event logs according to Definition 1, i.e., events are uniquely identified and can be associated with attributes. XL-IM stores the attributes of event logs outside of the main memory in a key-value store. Currently, XL-DB uses the embedded database MapDB as key-value store. MapDB is a disk-backed replacement of the Java collections framework. MapDB stores data in a BTree structure that is optimized for a low serialization overhead.

Similarly, to the storage layout of XL-IM the variant XL-DB views the event log as a table with n rows and m columns, where n is the number of events and m is the number of attributes. We store the values of all attributes together in a single key-value table. The unique key for each attribute value is determined by the tuple (x, y), where x is the unique identifier of the event (cf. Section 3.1) and y is a unique identifier generated for the attribute name. In contrast to XL-IM, we do not assume consistent universes for attributes. Therefore, we need to store meta-data on the possible values along with the value. For example, it would be possible that some values of attribute *granted* are stored as boolean values 1 and 0 and other values are stored as literals `true` and `false` .

The advantage over the XL-IM variant is that XL-DB uses MapDB as key-value store. MapDB transparently serializes values to a memory-mapped file on the disk. Therefore, the event log can exceed the amount of available memory.

However, we avoid the overhead associated with using a dedicated database server. By using memory-mapped files XL-DB can leverage all available memory of the machine, i.e., it is not constrained by the memory available to the Java Virtual Machine (JVM). The task of memory management is left to the operating system. The only structure left in the memory of the JVM for XL-DB are trace objects. Traces objects simply contain a list of event identifiers. Each event identifier occupies 8 bytes of memory that can be compressed as described in Section 3.1. Each trace object itself inflicts an overhead of 40 bytes. In addition, there is some constant overhead from the MapDB library itself that can be neglected for large event logs.

4 Benchmark

We compared the capabilities of all three MapDB variants against the naïve OpenXES implementation for several real-life event logs that are publicly available in the repository of the IEEE Taskforce on Process Mining. We also included an optimized variant of the OpenXES implementation, which adopted the flyweight pattern as proposed by XESLite (cf. Section 3.1). We report on the amount of memory used by the event log in the JVM (i.e., we do not count the memory allocated to MapDB by the operating system) and we report the run time of three basic operations:

A) loading the event log,
B) sequentially reading the activity name of all events, and
C) sequentially reading all attributes of all events.

4.1 Memory usage

Figure 6 shows the memory usage for 16 real-life event logs of different size. 13 of the events logs were obtained from the IEEE Taskforce on Process Mining repository [9,10,11,22,23,24,25,26]. The size of the event logs varies from 6,600 events [24], to more than 7 million events [10]. Three non-public event logs were included to have more large event logs with about 6 million events. Generally, the memory usage increases with the number of events in the event log. Moreover, as expected, event logs with a lower number of average attribute per event require less memory.

The OpenXES-Naïve and OpenXES-Flyweight variants use consistently more memory than any of the XESLite implementations. The effect of the flyweight optimization applied to literal values is clearly visible. Often, event logs contain only a small number of distinct literals, which are repeatedly used, e.g., the activity names. We could not load the two largest event logs with the OpenXES-Naïve variant with 8 GB of memory, therefore, the respective values are missing in Figure 6.

Both XL-DB and XL-IM variants use in the order of one magnitude less memory than their OpenXES counterparts. Theoretically, the disk-backed XL-DB variant should use less memory than XL-IM, which stores the whole event

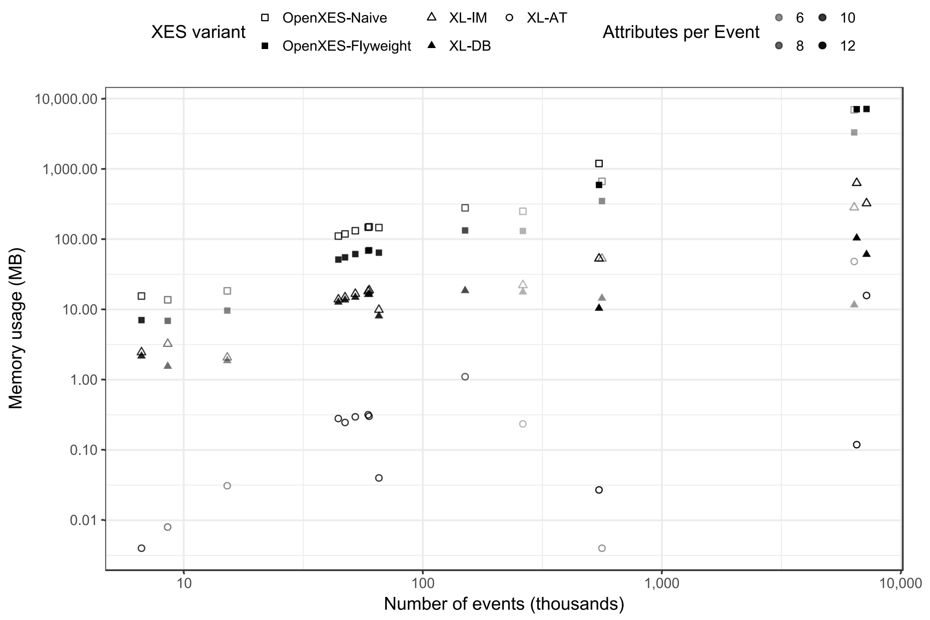

Fig. 6. Memory used by different XES implementations to store 16 real-life event logs. Darker data points depict logs with a higher average number of attribute per event.

log in-memory. Interestingly, for the majority of small event logs there is little difference in memory usage between XL-DB and XL-IM. Looking at those examples in more detail, we found there are two reasons for this observation: (1) some event logs (e.g., [11]) contain a lot of boolean attributes, which can be very efficiently stored in the XL-IM implementation; and (2) some event logs (e.g., [26]) contain a lot of literal attributes, which are kept in-memory due to the flyweight optimization for literals in XL-DB.

The XL-AT variant require the lowest amount of memory. However, it should be noted that this variant only stores the activity name of an event, i.e., the event identity and all other attributes are lost. Still, the memory required by the XL-AT variant gives a good estimate of the amount of information that is actual stored in the event log. The memory requirements of XL-AT are not linear to the number of events, but depend on how many distinct trace sequences are available and the similarity between the sequences. For example, XL-AT only requires 120 KB of memory to encode one of the non-public event logs with 8.2 million events. In contrast, more than 1 MB of memory is required to store the much smaller event log of the BPI challenge 2011 [22], which contains only 150,000 events.

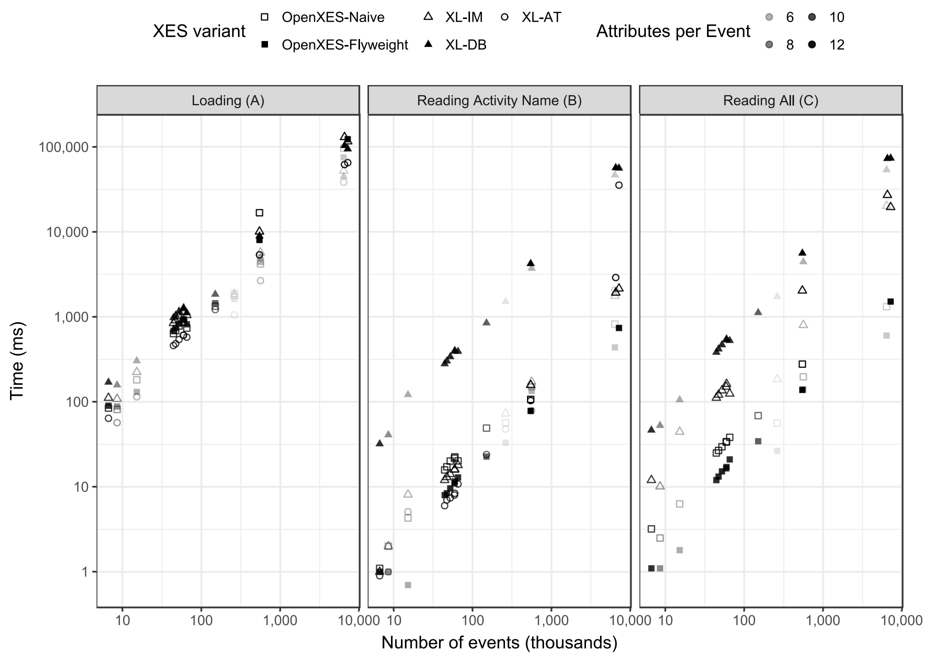

Fig. 7. Time required to execute three operations (A) loading the log, (B) sequentially reading the activity name of all events, and (C) sequentially reading all attributes of all events. Operation (C) is not measured for XL-AT. Darker data points depict logs with a higher average number of attribute per event.

4.2 Performance

Figure 6 shows the run-time of the three basic operations for all event logs under consideration. Clearly, the XL-DB variant is the slowest implementation for all three operations. Data needs to be stored on the hard-drive, which is considerably slower than the main memory. In one exceptional case, the OpenXES-Naïve variant required more time to load the event log than all other implementations. When looking at this case in more detail, we found that the additional time is spend by the JVM during excessive garbage collection. Here, the much lower memory footprint of the XESLite variants lead to a better performance. Regarding the remainder of the data, the differences between the loading times are less than one magnitude. In fact, most time during the loading of an event log is spent on parsing the time attributes and parsing the XML structure.

Interestingly, there seems to be little difference between reading all attribute (B) and reading only the activity name (C) when using XL-DB. Looking at the implementation, this can be explained by the fact that attributes of an event are stored close to each other. Internally, MapDB uses a BTree structure that loads full pages containing multiple records together into memory. Another notable observation is the high run time of operation B when using variant XL-AT for one specific event log. This outlier can be explained by the complex automaton that is build for this event log and an implementation detail in XL-AT. The current implementation mimics a random-access list with considerable overhead per request for larger automata. Instead it should sequentially iterate through all words of the automaton.

5 Conclusion

We systematically investigated the memory requirements for two types of event logs: normal event logs, in which events have unique identifiers and can be assigned multiple attributes, and simple event logs, which can be described by a multi-set of trace sequences without additional attributes. We identified the limits of the naïve implementation provided by the XES reference implementation OpenXES. We described XESLite, a software package that provides three alternative XES implementation: XL-AT, XL-IM, XL-DB, which can manage large event logs within the process mining framework ProM. We classified the suitability of the implementations provided by XESLite for event logs of different sizes and with different properties. For the variant XL-AT, we show that simple event logs can be very efficiently encoded with a deterministic acyclic finite automaton based on a perfect hashing scheme, which was originally developed for dictionaries. In this manner, even very large simple event logs can be stored in a small amount of memory. The variant XL-IM adopts the idea of column-based databases to densely pack together similar attributes. The variant XL-DB uses a flexible, embedded key-value store. Finally, we used a collection of 16 real-life event logs to conduct a benchmark of the performance of XESLite and compared to the performance of OpenXES. Our results show that XESLite can efficiently handle medium-sized to large event logs on standard work stations with limited available main memory without the need of external databases. Moreover, for control-flow discovery of a process model from a simple event log, XL-AT provides a very memory-efficient implementation, which can handle very large event logs based on an automaton.

References

1. IEEE Computational Intelligence Society: IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. (2016) IEEE Std 1849-2016.
2. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
3. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: CAiSE Forum. Volume 72 of Lecture Notes in Business Information Processing., Springer (2010) 60–75
4. Rosa, M.L., Reijers, H.A., van der Aalst, W.M.P., Dijkman, R.M., Mendling, J., Dumas, M., García-Bañuelos, L.: APROMORE: an advanced process model repository. Expert Syst. Appl. **38**(6) (2011) 7029–7040

5. Günther, C.W.: Process mining in flexible environments. PhD thesis, Technische Universiteit Eindhoven (2009)
6. van Dongen, B.F., Shabani, S.: Relational XES: data management for process mining. In: CAiSE 2015 Forum. Volume 1367 of CEUR Workshop Proceedings., CEUR-WS.org (2015) 169–176
7. Syamsiyah, A., van Dongen, B.F., van der Aalst, W.M.P.: DB-XES: enabling process discovery in the large. In: SIMPDA 2016. Volume 1757 of CEUR Workshop Proceedings., CEUR-WS.org (2016) 63–77
8. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: CAiSE. Volume 9694 of Lecture Notes in Computer Science., Springer (2016) 290–305
9. de Leoni, M., Mannhardt, F.: Road Traffic Fine Management Process. Eindhoven University of Technology. Dataset (2015) doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.
10. Dees, M., van Dongen, B.: BPI Challenge 2016. UWV. Dataset (2016) doi:10.4121/uuid:360795c8-1dd6-4a5b-a443-185001076eab.
11. Mannhardt, F.: Sepsis Cases - Event Log. Eindhoven University of Technology. Dataset (2016) doi:10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.
12. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: Petri Nets. Volume 7927 of Lecture Notes in Computer Science., Springer (2013) 311–329
13. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: CIDM, IEEE (2011) 310–317
14. Günther, C.W., van der Aalst, W.M.P.: Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 328–343
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
16. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Software: Practice and Experience **45**(1) (2015) 1–29
17. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition (2. ed). Addison-Wesley (2003)
18. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.: Incremental construction of minimal acyclic finite state automata. Computational Linguistics **26**(1) (2000) 3–16
19. Lucchesi, C.L., Kowaltowski, T.: Applications of finite automata representing large vocabularies. Softw., Pract. Exper. **23**(1) (1993) 15–30
20. Daciuk, J., Maurel, D., Savary, A.: Dynamic perfect hashing with finite-state automata. In: Intelligent Information Systems. Advances in Soft Computing, Springer (2005) 169–178
21. Abadi, D.J., Madden, S., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOD Conference, ACM (2008) 967–980
22. van Dongen, B.: Real-life event logs - Hospital log. Eindhoven University of Technology. Dataset (2012) doi:10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54.
23. van Dongen, B.: BPI Challenge 2012. Eindhoven University of Technology. Dataset (2012) doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.
24. Steeman, W.: BPI Challenge 2013. Ghent University. Dataset (2013) doi:10.4121/uuid:a7ce5c55-03a7-4583-b855-98b86e1a2b07.

25. Buijs, J.: Environmental permit application process ('WABO'), CoSeLoG project. Eindhoven University of Technology. Dataset (2014) doi:10.4121/uuid:26aba40d-8b2d-435b-b5af-6d4bfbd7a270.
26. van Dongen, B.: BPI Challenge 2015. Eindhoven University of Technology. Dataset (2015) doi:10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.