Mavroudopoulos *et al. Process Science (2025) 2:6*
https://doi.org/10.1007/s44311-025-00013-9

Process Science

RESEARCH Open Access

Declarative process mining in big data scenarios using an application-agnostic framework

Ioannis Mavroudopoulos[1], Christos Balaktsis[1], Konstantinos Varvoutas[1], Georgia Kougka[1], Anastasios Gounaris[1] and Marco Comuzzi[2*]

*Correspondence:
mcomuzzi@unist.ac.kr

[1] Department of Computer Science, Aristotle University of Thessaloniki, Thessaloniki, Greece
[2] Department of Industrial Engineering, Ulsan National Institute of Science and Technology, Ulsan, South Korea

Abstract

Although they are usually designed for top-notch scalability and flexibility, application-agnostic big data pattern analysis frameworks are seldom exploited in process mining. As the size of event logs and the velocity at which events can be generated grow, however, the need for big data-aware process mining solutions emerges. This work targets the extraction of declarative process constraints. Its key novelty lies in employing an application-agnostic pattern analysis framework, called SIESTA, rather than devising an ad-hoc solution tailored to declarative constraint discovery. The key contribution of our work is threefold: (i) we show how we can build on top of SIESTA to extract the full set of Declare constraints in large event logs in a more efficient and scalable manner than the ad-hoc competitors; (ii) we extend our SIESTA-based approach to operate in an incremental manner, which may be required both when the event logs are very large and when they are continuously updated by new batches of events; and (iii) we demonstrate how our SIESTA-based framework can be extended to mine temporal violations of Declare constraints to support variant analysis. The experimental results show that our solution can ingest and process thousands of events per second even using a commodity machine, it can handle datasets of tens of millions of events, and it is much faster than the competitors in repetitive constraint extraction tasks for larger datasets than the ones that can be handled by the competitors.

Keywords: Business process constraints, Process mining, Big data, Data streams, Temporal violations

Introduction

Process mining (PM) is a discipline within Business Process Management (BPM) that aims at extracting useful knowledge from the so-called *event logs*, that is, logs of the information systems that support the business process (BP) execution (Dumas et al. 2018). The principal use case of PM is process discovery, which aims at obtaining a model of the control flow of a process, capturing the order in which the activities in an event log are usually executed. Imperative process models (van der Aalst and Carmona 2022) aim at capturing explicitly all the control flow relationships among classes of activities. In scenarios characterized by high variability, such as diagnosis and treatment in

healthcare, declarative process models (Maggi et al. 2012) are often more appropriate. These take the form of constraints, also referred to as patterns or rules,[1] on the order of activities in the process execution, e.g., activity *b* cannot happen after activity *a*. Within the boundaries set by those constraints, process execution can be left free.

Process discovery in declarative process modelling turns into extracting declarative constraints among activities in an event log (Maggi et al. 2012). Declare (Pesic et al. 2007) is one of the languages that can be used to express such constraints. Extracting Declare constraints can become a computationally intensive task, and particularly challenging when it has to be repeated multiple times on the same data and/or in a short period. This happens, for instance, when it is also necessary to calculate the support of such constraints, or when the event log data are delivered as a stream or in batches. In the former case, intuitively, discovered Declare constraint templates must be checked against every trace in a log to verify their support. The latter case is a scenario that generally poses challenges in (big) data processing, since the Declare constraint templates discovered should be updated when new event log data become available.

The methods to extract process constraints from event logs that have appeared in the literature thus far can be classified as ad-hoc (or *application-specific*), i.e., they are developed with one specific constraint extraction target in mind, such as mining the full set of Declare constraints (Augusto et al. 2021; Maggi et al. 2018), adding activation conditions on data payloads to such constraints (Maggi et al. 2013), or mining the constraints in the event log streaming setting (Burattin et al. 2015). During the past ten years, however, the big data research has proposed several general-purpose (or *application-agnostic*) big data pattern analysis (BDPA) frameworks, e.g. (Ziehn et al. 2024), that can be adapted with limited customisation to different scenarios. Constraint extraction from event logs can be one of such scenarios where general-purpose BDPA can be employed effectively.

More specifically, we argue that general-purpose BDPA frameworks can better serve the purpose of constraint extraction from event logs in two ways:

- *Increasing performance*: general-purpose BDPA frameworks are usually leveraged in data-intensive scenarios, characterized by the need to process (very) large amounts of data, possibly continuously delivered, where performance becomes critical. In the specific case of business process constraints extraction, one such example scenario occurs when the constraint extraction must be repeated multiple times for different constraint support threshold values. We argue that, in data-intensive scenarios, general purpose BDPA, with limited adaptations, could exceed the performance of ad-hoc solutions.
- *Enhancing flexibility*: while stand-alone methods are tailored to specific settings, general-purpose BDPA frameworks are usually easy to customise in order to be adapted to different constraint extraction scenarios with minimal adaptations. Therefore, we argue that even in the context of constraint extraction from event logs, general-purpose BDPA can be exploited to address constraint scenarios that are not addressed by ad-hoc solutions, and for solutions, which the current ad-hoc solutions could not be easily adapted.

[1] In this work, we use the terms pattern, constraint, and rule interchangeably.

To demonstrate the claims that we made above, in this paper we exploit a general-purpose BDPA framework recently published named SIESTA (Mavroudopoulos and Gounaris 2024, 2022), a scalable system specifically designed for efficient pattern analysis over large log files. Its efficiency relies on building inverted indices of continuously arriving logs, which can then speedup pattern query execution. We first demonstrate that, with limited extensions, SIESTA can be seamlessly adapted to perform Declare constraint extraction from event logs. Then, we demonstrate that SIESTA outperforms all the existing ad-hoc solutions in the literature in data-intensive Declare constraint extraction scenarios, i.e., when the extraction must be repeated multiple times for different support thresholds or the dataset is very large. Finally, we show how SIESTA can be extended to flexibly support two scenarios. Regarding data delivery, we show how SIESTA can efficiently support the scenarios in which logs are continuously delivered in batches. Regarding the type of constraint extracted, we show how SIESTA can be extended to perform temporal variant analysis based on the assessment of temporal constraint violations.

The results of our approach are promising. We efficiently extracted the complete set of Declare patterns from datasets consisting of up to 96M events using a commodity machine, while none of our competitors, such as the Declare Miner (Maggi et al. 2012), used by RuM (Alman et al. 2020) and MINERful (Di Ciccio and Mecella 2015), could complete such a task. Our solution exhibits throughput of several thousand events per second. The performance advantage of our solution becomes even more pronounced when the extraction process is executed multiple times with varying support thresholds, which is when the indices of our solution are better utilized. In such scenarios, our incremental solution is up to 2.82 faster than the best-performing competitor.

In summary, our contribution is threefold: (i) we show how we can build on top of SIESTA to extract the full set of Declare constraints in a more efficient and scalable manner than competitors in big datasets; (ii) we extend the previous results to operate in an incremental manner, which is required both when event logs are very large and when logs are analyzed while they are updated; and (iii) we explain how we can perform advanced temporal variant analysis using the same framework by focusing on temporal constraint violations. Our work also conveys a broader message, that is, we advocate for leveraging existing general-purpose big-data analysis frameworks for declarative process mining tasks, as opposed to developing ad-hoc solutions for each specific task. Our implementation is publicly available.[2]

This work is an extension of the work presented in Mavroudopoulos et al. (2024). Compared to our previous publication, this paper provides: (i) an extended new set of experiments about the basic Declare constraint extraction scenario, focusing in particular on the scalability of our proposed approach; (ii) a first extension of SIESTA to support Declare constraint extraction when event logs are delivered in batches; and (iii) a second extension of SIESTA supporting the mining of temporal Declare constraint violations.

[2] https://github.com/mavroudo/SequenceDetectionQueryExecutor/releases/tag/v3.0.0 for the Declare constraint miner, https://github.com/siesta-tool/DeclareIncremental/releases/tag/v1.0.0 for the extensions introduced in this paper, and https://github.com/siesta-tool/siesta-demo for the complete system and for replicating all the experiments discussed in the paper.

The remainder is structured as follows: Related work and Preliminaries sections present the related work and the background, respectively. The first part of the contribution, showcasing how we leverage SIESTA's indices for efficient extraction of the Declare patterns, in both a batch and an incremental manner, is presented in Mining declarative constraints section. Temporal variant analysis section explains how we can leverage our framework to detect temporal Declare constraint violations and thus perform advanced variant analysis. Evaluation section reports on the performance evaluation, while we conclude in Conclusion and future work section.

Related work

Table 1 lists, in chronological order, the contributions in the literature aiming at discovering different types of declarative process constraints. All the early contributions in the literature focus on extracting Declare constraints (the detailed list of the constraints in the Declare language is provided in Table 2). The first approach (Maggi et al. 2012) proposes a two-phase procedure where the Apriori algorithm is first used to generate a list of candidate Declare constraints. These are further pruned based on relevance using

Table 1 Summary of related work on declarative constraints discovery (CTL: computational tree logic, CRL: compliance request language, DCR: dynamic condition response graphs)

Ref.(year)	Language	Type of Constraints	Notes
Maggi et al. (2012)	Declare	Control-flow	First approach for constraint discovery based on Apriori algorithm
Westergaard and Stahl (2013)	Declare	Control-flow	UncostrainedMiner: Fast discovery of constraints with no predefined assumption about constraints
Maggi et al. (2013)	Declare	Control-flow, event data	Discover Declare constraints with data conditions on constraint activations
Di Ciccio and Mecella (2015)	Declare	Control-flow	MINERful: Best-in-class discovery algorithm based on a two-phase technique
Burattin et al. (2015)	Declare	Control-flow	First approach to discover constraints in event streams
Elgammal et al. (2016)	CRL	Control-flow, time, resources	Discover CRL constraints satisfied by all traces in an event log
Maggi et al. (2018)	Declare	Control-flow	First parallel implementations of different constraint discovery algorithms
Leno et al. (2018)	Declare	Control-flow, event data	Extend (Maggi et al. 2013) with data conditions on constraint target conditions
van Beest et al. (2019)	CTL	Control-flow	Discover control-flow process variants that are satisfied by all traces in an event log
Navarin et al. (2020)	Declare	Control-flow, event data	First approach to discover data-aware Declare constraints in event streams
Alman et al. (2020)	Declare	Many	RuM: Stand-alone toolkit implementing several discovery techniques for Declare constraints
Back et al. (2022)	DCR	Control-flow, time, resources	Discover dynamic condition response constraints satisfied by all traces in an event log
Donadello et al. (2022)	Declare	Many	Declare4Py: Python package implementing several discovery techniques for Declare constraints

Table 2 Declare patterns

Declare pattern	Formal Definition	Description
Existence Patterns		
existence(*a*,*n*)	$\|O_a^\sigma\| \geq n$	*a* exists at least *n* times in σ
absence(*a*,*n*)	$\|O_a^\sigma\| \leq n-1$	*a* exists at most $n-1$ times
exactly(*a*,*n*)	$\|O_a^\sigma\| = n$	*a* exists exactly *n* times
Unordered Relation Patterns		
co-existence(*a*,*b*)	$(\|O_a^\sigma\| \geq 1 \wedge \|O_b^\sigma\| \geq 1) \vee (\|O_a^\sigma\| = 0 \wedge \|O_b^\sigma\| = 0)$	if *a* occurs, then *b* occurs and vice versa
not co-existence(*a*,*b*)	$(\|O_a^\sigma\| \geq 1 \wedge \|O_b^\sigma\| = 0) \vee (\|O_a^\sigma\| = 0 \wedge \|O_b^\sigma\| \geq 1) \vee (\|O_a^\sigma\| = 0 \wedge \|O_b^\sigma\| = 0)$	*a* and *b* never occur together
choice(*a*,*b*)	$\|O_a^\sigma\| \geq 1 \vee \|O_b^\sigma\| \geq 1$	at least one of *a* or *b* occurs
exclusive choice(*a*,*b*)	$(\|O_a^\sigma\| \geq 1 \wedge \|O_b^\sigma\| = 0) \vee (\|O_a^\sigma\| = 0 \wedge \|O_b^\sigma\| \geq 1)$	one of *a* or *b* occurs, but not both
responded existence(*a*,*b*)	$\|O_a^\sigma\| \geq 1 \Rightarrow \|O_b^\sigma\| \geq 1$	if *a* occurs *b* occurs as well
Position Patterns		
init(*a*)	$ev_1.type = a$	first event in σ is *a*
last(*a*)	$ev_n.type = a$	last event in σ is *a*
Ordered Relation Patterns		
response(*a*,*b*)	$\forall ev_i \| ev_i.type = a \; \exists ev_j, i < j \| ev_j.type = b$	if *a* occurs, then *b* occurs after *a*
precedence(*a*,*b*)	$\forall ev_j \| ev_j.type = b \; \exists ev_i, i < j \| ev_i.type = a$	*b* occurs only if preceded by *a*
succession(*a*,*b*)	response(*a*,*b*)$\wedge$ precedence(*a*,*b*)	*a* occurs if and only if it is followed by *b*
not succession(*a*,*b*)	$\forall ev_i \| ev_i.type = a \;\; \nexists ev_j, i < j \| ev_j.type = b$	*a* can never occur before *b*
alternate response(*a*,*b*)	$\forall ev_i \| ev_i.type = a \; \exists ev_j, i < j \| ev_j.type = b \wedge \forall ev_k, i < k < j \| ev_k.type \neq a$	Each time *a* occurs, then *b* occurs afterwards, before *a* may recur
alternate precedence(*a*,*b*)	$\forall ev_j \| ev_j.type = b \; \exists ev_i, i < j \| ev_i.type = a \wedge \forall ev_k, i < k < j \| ev_k.type \neq b$	Each time *b* occurs, it is preceded by *a* and no other *b* can recur in between
alternate succession(*a*,*b*)	alternate response(*a*,*b*)$\wedge$ alternate precedence(*a*,*b*)	*a* and *b* occur in pairs
chain response(*a*,*b*)	$\forall ev_i \| ev_i.type = a, ev_{i+1}.type = b, i \in [0, \|\sigma\| - 1]$	Each time *a* occurs, then *b* occurs immediately afterwards
chain precedence(*a*,*b*)	$\forall ev_i \| ev_i.type = b, ev_{i-1}.type = a, i \in [1, \|\sigma\|]$	Each time *b* occurs, then *a* occurs immediately beforehand
chain succession(*a*,*b*)	chain response(*a*,*b*)$\wedge$chain precedence(*a*,*b*)	*a* and *b* appear concecutive in pairs
not chain succession(*a*,*b*)	$\exists ev_i \| ev_i.type = a \wedge \; ev_{i+1}.type \neq b$	*a* is never immediately followed by *b*

various metrics, including confidence, support, interest factor, and so on. This approach is similar to our work in that we both focus on search space reduction. However, our solution emphasizes performance issues.

The tool UnconstrainedMiner (Westergaard and Stahl 2013) implements fast and accurate mining of Declare constraints without requiring predefined assumptions about the model. This tool manages to handle large logs efficiently in parallel and provides all constraints for post-processing to transform event data into a structured format for immediate constraint mining and the addition of new constraints based on their LTL semantics. However, its performance is exceeded by later contributions, such as MINERful (Di Ciccio and Mecella 2015).

The early approach of Maggi et al. (2012) has been then extended along the two directions that we consider in this paper to extend our previous work: the continuous delivery of events, i.e., the streaming setting (Burattin et al. 2015), and the specification of temporal conditions, as a specific class of data-aware Declare constraints (Maggi et al. 2013).

The solution for streaming settings proposed in Burattin et al. (2015) considers a continuous stream of events that are processed individually and relies on typical techniques for streaming data processing, such as sliding windows and lossy counting, which implies approximate processing. In this paper, we do not consider the full-fledged streaming setting, but the delivery of events in batches. However, we guarantee exact results and our framework does not make assumptions regarding practical aspects, such as clear indication of start and end events in each case, as Burattin et al. (2015) does.

The MINERful (Di Ciccio and Mecella 2015) algorithm discovers the basic set of constraints, showing remarkable performance. This is also a two-phase technique, where the first phase produces statistical information extracted from event logs, while the second one discovers the constraints and filters them based on the user-defined thresholds. MINERful stores its indices in-memory, which makes it challenging to update them across multiple batches of events. As a result, the only viable approach for handling incremental updates is to re-execute the process from scratch, incorporating all previous events.

An approach that extends the rationale of Maggi et al. (2012) for parallel discovery of declarative process constraints is introduced by Maggi et al. (2018) and aims to improve the time performance exploiting concurrency. The parallelization is achieved by two different partitioning methods: search space and database partitioning. Note that SIESTA, on which our proposal relies, is scalable by design and capable of benefiting from massive parallelism through its implementation in Spark and the underlying data storage layer.

The Declare Miner plug-in of the ProM toolkit (van Dongen et al. 2005) implements several discovery algorithms for Declare constraints. Both the Declare Miner and MINERful have been implemented in the Rule Mining toolkit RuM (Alman et al. 2020). The Declare4Py Python package (Donadello et al. 2022) provides programmatic access to several declarative process modelling techniques, but without an explicit focus on performance, particularly as far as declarative constraints discovery is concerned.

Declare constraints can be extended with Boolean conditions on the activation and target activities, resulting in the so-called data-aware class of Declare constraints. The discovery of data-aware Declare constraints has been addressed originally by Maggi et al. (2013) on the activation conditions of the constraints and extended to the target conditions in Leno et al. (2018), and to the streaming setting in Navarin et al. (2020). The basic approach for data-aware constraint discovery is based on combining frequent pattern analysis for discovery the constraints with clustering techniques to discern the relevant data conditions.

Other approaches, such as van Beest et al. (2019) and DisCoveR (Back et al. 2022), also discover declarative constraints similar to those described in the Declare language, focusing on patterns that are supported in all the traces of an event log. Unlike these approaches, our method does not focus only on discovered patterns that appear in all traces in a log, as the discovered patterns depend on a user-defined support threshold. Specifically, van Beest et al. (2019) aims to extract patterns that describe all the possible execution paths of a business process, resulting in rules with multiple alternatives, e.g., “activity A is followed by B or C". DisCoveR extracts patterns, modeled as Dynamic Condition Response (DCR) graphs, that hold true in all traces of an event log. Therefore,

these approaches normally yield a smaller set of patterns compared to our solution. Regarding expressivity, the method in van Beest et al. (2019) considers neither most unordered, existence, and position templates nor the alternate ones. However, it does capture the cyclic and concurrency relationships, which can only be implicitly deduced by Declare patterns (e.g., concurrency can be inferred by the co-existence of two activities without a specified order). DisCoveR supports most Declare constraints but lacks templates like responded existence and chain precedence. Additionally, both DCR and the Compliance Request Language (CRL) (Elgammal et al. 2016) can describe constraints based on attributes such as time and resources, which SIESTA currently does not support. Note that our SIESTA-based solution, through its integration with a CEP engine, can detect any pattern expressible as a regular expression of activities without nested expressions.

It is important to note that there is another set of open-source implementations (Schönig et al. 2016; Zaki et al. 2022; Augusto et al. 2021; Esser and Fahland 2021) that perform process mining and pattern extraction from event logs using SQL or graph-based database architectures and technologies. The approach by Schönig et al. (2016) applies SQL queries to relational event data, while Esser and Fahland (2021) propose a novel data model for more efficient querying and event data transformations using Neo4j's Cypher graph query language. A similar interesting research direction is proposed by Zaki et al. (2022), where the encoding of event logs as graphs in Neo4j is adopted for compliance checking. Additionally, the compliance rule evaluation by pre-defining the data structure is addressed by Augusto et al. (2021), where the recent MATCH_RECOGNIZE SQL extension is shown to be the dominant solution. In our earlier work in Mavroudopoulos et al. (2024), we compared our proposed solution against both Neo4j and MATCH_RECOGNIZE and provided strong evidence that such solutions are inferior.

In summary, there are no contributions in the literature that focus on extracting declarative constraints efficiently using existing BDPA frameworks. More specifically, the temporal violations of constraints that we consider in this extension of our previous work can be seen as a specific class of data-aware constraints where conditions are expressed only over timestamps. While the approaches in the literature aim at discovering all types of data-aware constraints, without focusing specifically on performance, in this work we use temporal constraint violations as a means to show how our SIESTA-based approach to Declare constraint discovery can be easily extended to accommodate data-aware conditions. Note that both SIESTA (Mavroudopoulos and Gounaris 2024) and proposals such as Augusto et al. (2021) or MATCH RECOGNIZE inherently support temporal constraint extraction. However, the problem of detecting temporal constraint violation has not been addressed before in declarative process discovery.

Preliminaries

SIESTA's basics

We begin with a brief description of the notation and an overview of the SIESTA system along with the proposed indexing extension to support the extraction of Declare constraints (described in Declare constraints section). SIESTA is an infrastructure for efficient support of sequential pattern queries based on inverted indices, generated from

a logfile $\mathcal{L}$ containing timestamped events E. The events are of a specific type and are logically grouped into sets called cases, sessions, or traces. More formally:

Definition 1 (Event Log) Let A be a finite set of activities (aka tasks). A log $\mathcal{L}$ is defined as $\mathcal{L} = (E, C, \gamma, \delta, \prec)$, where E is the finite set of events, C is the finite set of Cases, $\gamma : E \rightarrow C$ is a function assigning events to Cases, $\delta : E \rightarrow A$ is a function assigning events to activities. An event $ev \in E$ that belongs in a case $\sigma \in C$, is a tuple that consists of at least a recorded timestamp ts, denoting the recording of task execution, an event type $type \in A$ (i.e., $\delta(ev) = ev.type$), and a position pos, denoting the position of the event in σ (i.e., $\sigma = \langle ev_1, ev_2, \ldots, ev_n \rangle$ and $ev_i.pos = i$). Finally, $\prec$ is the strict total ordering over events belonging to a specific case, deriving from the event timestamps.

According to the above definition, the event timestamps provide an ordering between the events that belong in a trace and SIESTA requires a clear ordering between the events in each trace to accurately create its indices. Therefore, we assume that two events within the same trace cannot have the same timestamp. As evident from real-world event logs, the timestamp of each event typically corresponds to either the beginning of the activity or its completion. Although event logs may also contain timestamps related to lifecycle information, e.g., when activity is enabled or paused, these are not required by our implementation.

Below, we provide additional definitions required by SIESTA:

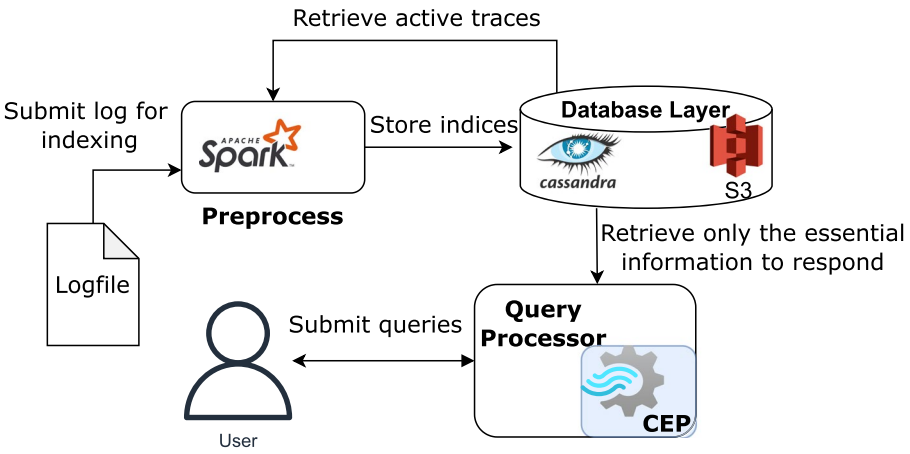
Definition 2 (*et-pair*). Given a set of activities (or tasks) A, an event type pair, or *et-pair* for short, is a pair (a_i, a_j), where $a_i, a_j \in A$.

Definition 3 (*event-pair*). For a given *et-pair* (a_i, a_j) and a sequence of events $\sigma = \langle ev_1, ev_2, ..., ev_n \rangle$, there is an *event-pair* (ev_x, ev_y), where $ev_x, ev_y \in \sigma$, if $x < y \ \wedge \ ev_x.type = a_i \ \wedge \ ev_y.type = a_j$. Note that based on this definition, the events in an *event-pair* do not have to be consecutive.

Definition 4 (*event-pairs non-overlapping in time*). Two *event-pairs* (ev_x, ev_y) and (ev_i, ev_j) are non-overlapping in time if $x > j \vee y < i$.

In the original SIESTA (Mavroudopoulos and Gounaris 2022), *event-pairs* are generated based on Definition 4. However, in recent work (Mavroudopoulos and Gounaris 2024), SIESTA is extended to allow two *event-pairs* to partially overlap in time. This change affects only the *event-pairs* (ev_1, ev_2) where $ev_1.type = ev_2.type$ (i.e., both event types are the same), in which case the second event can be reused as the first event in another *event-pair* instance. This modification ensures that the list containing *event-pairs* (ev_1, ev_2), with $ev_1.type = ev_2.type = a_i$, includes all instances of a_i in $\mathcal{L}$, which is utilized for efficiently retrieving relevant events when calculating Declare constraints, as discussed in the next section.

SIESTA's architecture (illustrated in Fig. 1a) consists of the pre-processing component and the query processor, along with the database layer. The pre-processing component is responsible for handling continuously arriving logs and computing the appropriate indices. After the indices are computed, they are stored in a database. In



(a) Architecture

Logfile		
EventType	**Trace ID**	**Timestamp**
A	t1	2024-06-02
A	t1	2024-06-03
A	t2	2024-06-03
B	t1	2024-06-08
B	t2	2024-06-09
C	t2	2024-06-10

available et-pairs
(A,A) (A,B) (A,C) (B,C)

Indices

IndexTable
(A, A) -> [(t1, 1, 2)]
(A, B) -> [(t1, 1, 3), (t2, 1, 2)]
(A, C) -> [(t2, 1, 3)]
(B, C) -> [(t2, 2, 3)]

SingleTable
(A) -> [(t1, 1), (t1, 2), (t2, 1)]
(B) -> [(t1, 3), (t2, 2)]
(C) -> [(t2, 3)]

LastChecked
(A,A,t1) -> 2024-06-03
(A,B,t1) -> 2024-06-08
(A,B,t2) -> 2024-06-09
(A,C,t2) -> 2024-06-10
(B,C,t2) -> 2024-06-10

CountTable
(A, A) -> (1 occurence, 1 day)
(A, B) -> (2 occurrences, 6 days)
(A, C) -> (1 occurrence, 7 days)
(B, C) -> (1 occurrence, 1 day)

SequenceTable
(t1) -> [(A, 2024-06-02, 1), (A, 2024-06-03, 2), (B, 2024-06-08,3)]
(t2) -> [(A, 2024-06-03, 1), (B, 2024-06-09, 2), (C, 2024-06-10,3)]

(b) Indexing example

Fig. 1 System overview

our implementation, we use S3, though Apache Cassandra[3] can serve as an alternative. Finally, the query processor utilizes the stored indices to perform efficient pattern analysis and integrates a Complex Event Processing (CEP) engine to perform efficient pattern

[3] https://cassandra.apache.org/_/index.html

analysis. Later, we will explain how we extended this architecture to efficiently support incremental declarative process mining.

The primary inverted index, named *IndexTable*, is created based on the *et-pairs*. For each *et-pair*, a list of non-overlapping (partially overlapping when both activities in the *et-pair* are the same) *event-pairs* is extracted from each trace. These lists are then combined into a single list. The relevant information stored for every *event-pair* is the trace id σ_{id} in which it was detected and the timestamps of the two events in the pair. Besides the *IndexTable*, there are additional auxiliary tables that enable different processes. The most relevant one to our work is the *SingleTable*, which stores records with an event type as key and a value containing all occurrences of this event type in the log. The remaining tables are: *SequenceTable*, *CountTable*, and *LastChecked*. *SequenceTable* stores, for each event, its trace identifier, timestamp, and position in the trace. Except for the position (which requires grouping events by trace), the data it contains mirrors the raw log file. *CountTable* holds basic statistics for every available *et-pair* (i.e., an *et-pair* for which at least one *event-pair* exists in $\mathcal{L}$), such as the number of occurrences and the cumulative timespan between the two events. Finally, *LastChecked* stores the last timestamp for each event type per trace.

The discussion above refers to the logical deign level. At the physical level, we slightly modify the structure of the tables stored in S3 to address a limitation that otherwise would impact on the performance of incremental index building. Specifically, S3 stores indices in parquet[4] files, which are immutable. As a result, retaining inverted lists would require recreating them each time a new batch was processed. To avoid this, in practice, we create a record for each event pair in the *IndexTable* and partition these records by the event types. This allows us to preserve the efficient filtering while enabling new pairs to be appended. A similar approach is applied to the auxiliary tables, namely *SingleTable*, *SequenceTabl*e, and *LastChecked* (used to avoid generating pairs that have already been stored). The updated preprocessing code is publicly available in the provided repository.

Example: Let us consider a web-logging application that monitors the order of different actions denoted as A, B, and C, for two users. Figure 1b presents this log file along with the corresponding available *et-pairs* and SIESTA's built indices. After calculating and storing the indices, the query processor, implemented to operate in parallel, efficiently utilizes them to respond to various query types. The supported query types are: (i) *Pattern Detection:* This query returns all traces containing the specified pattern (e.g., $\langle C, B, A \rangle$); (ii) *Statistics:* This query provides basic statistics for an event-type pair (*et-pair*); and (iii) *Pattern Continuation:* This query identifies events most likely to extend the queried pattern.

In this work, we extend the supported queries by adding two additional functionalities: (i) declarative business process constraint extraction and (ii) differentiation between log partitions according to temporal constraints.

[4] https://parquet.apache.org/

Declare constraints

The patterns in Declare fall into three distinct categories: existence and unordered relation, position, and ordered relations (Maggi et al. 2011; De Smedt et al. 2019). Table 2 presents various Declare patterns that can be applied to a sequence $\sigma = \langle ev_1, \ldots, ev_n \rangle$, considering the event types a and b (i.e., $a, b \in A$) (Di Ciccio and Mecella 2015). This table employs the following definition below:

Definition 5 (Event type occurrences) For a trace $\sigma = \langle ev_1, ev_2, \ldots, ev_m \rangle$, the list of occurrences of $a \in A$ in σ are denoted as $O_a^\sigma = [ev_i | ev_i \in \sigma \wedge ev_i.type = a]$.

Mining declarative constraints

This section is split into two parts. The first ones describes the main contribution of our previous work (Mavroudopoulos et al. 2024). The second one presents our novel approach to incremental Declare constraint mining.

Mining declare constraints on top of SIESTA

In this part, we show how we can efficiently extract declarative process modelling constraints and more specifically, we show that we can cover all constraint templates (i.e., patterns) defined in the Declare modelling approach (Maggi et al. 2011; De Smedt et al. 2019).

We now focus on SIESTA's query processing extension to support the efficient extraction of Declare patterns. In our implementation, we allow the support threshold to be defined during querying, enabling the extraction of patterns with varying support values until the optimal threshold is discovered. This threshold dictates the minimum support that each pattern (i.e., constraint template) must reach to be included in the result set. For existence, unordered relations, and position patterns, we calculate support as the proportion of traces that validate the pattern. However, for ordered relation patterns, support is akin to confidence in data mining (i.e., the number of rule fulfilment divided by the total number of rule activations). Moreover, to facilitate the efficient extraction of all Declare patterns together based on a single threshold, we group relevant patterns according to the required information and execute them in a single run. This approach allows us to optimize the process, as costly operations, such as fetching data from the indices and joining records, are executed only once.

Ordered relation patterns

As discussed in Preliminaries section, the process of detecting *event-pairs* in SIESTA during index building allows for the skipping of irrelevant events in between, which is a requirement to support all Declare patterns. Consider the scenario where a more strict approach was implemented, where *event-pairs* are generated only between consecutive events (like in directly-follows graphs); then, from the ordered relation patterns, only the chain patterns would have been extractable. The IndexTable and SingleTable contain all the necessary information for extracting the ordered relation patterns. Algorithm 1 provides an abstract outline of this process. It takes the type of the pattern as input (e.g., alternate response) and returns the detected ordered relation patterns.

Algorithm 1 Extract ordered relation patterns

Input *pattern-type*, *support*
Output *Ordered Relation Patterns*
1: **for every** (a,b) **in** IndexTable **do**
2: ut[a,b] ← **extract_unique_traces**(IndexTable[a,b])
3: **for every** (a,b) **in** ut **do**
4: temp ← SingleTable[a].keyBy(σ_{id}) ⋈ SingleTable[b].keyBy(σ_{id})
5: R[a,b] ← temp.filterById(ut[a,b]) ▷ R[a,b] = [(σ_{id},O^{σ}_{a},O^{σ}_{b}),...]
6: C ← **count_occurrences**(R, *pattern-type*)
7: X ← **extract_total_occurrences**(SingleTable)
8: **return filter**(C, X, *support*, *pattern-type*)

For an *et-pair* (a,b), the IndexTable[a,b] stores all non-overlapping occurrences of this pair, as defined in Definition 4. However, a trace may contain multiple instances of both event types a and b. For example, consider the trace $\langle a_1, a_2, b_1, b_2\rangle$. According to the event pair extraction process described in Preliminaries section, only the pair $\langle a_1, b_1\rangle$ is stored in the IndexTable, as $\langle a_2, b_1\rangle$ or $\langle a_2, b_2\rangle$ would overlap in time with $\langle a_1, b_1\rangle$.

When counting instances where event b occurs after event a (i.e., response(a,b)), all occurrences of both a and b in the trace must be considered. To achieve this, we first extract the unique traces *ut* that contain at least one instance of the *et-pair* (a,b) (lines 1–2). We then join the occurrences of a and b within the same trace using the SingleTable (line 4). Alternatively, IndexTable[a,a] (resp. IndexTable[b,b]) could also be utilized. For instance, consider the trace $\langle a_1, a_2, b_1, a_3\rangle$. After applying the modifications discussed in the previous section, IndexTable[a,a] will include the event pairs (a_1, a_2) and (a_2, a_3), where a_2 is part of both pairs. Consequently, retrieving IndexTable[a,a] provides access to all occurrences of event type a. Since both IndexTable[a,a] and SingleTable[a] yield the same results, we opt for the one with the smaller overall size, which depends on the case.

The algorithm then retains only the traces that appeared in the extracted unique traces *ut* (line 5). After joining these records, we obtain all occurrences of both event types within the traces that contain at least one instance of the pair (a,b). The resulting structure, denoted as R[a,b], has the following format: $[(\sigma_{id}, O^{\sigma}_{a}, O^{\sigma}_{b}), \dots]$, where O^{σ}_{x} is a list of all occurrences of event type x in trace σ, with σ_{id} representing the trace identifier (Definition 5).

Simple Patterns. Now that all the relevant information is present, we can proceed to counting the total occurrences of each pattern. This process takes place in the `count_occurrences` function (line 6). The function operates as follows: for a pair (a, b), we iterate through O^{σ}_{a} and count how many of these events have an event in O^{σ}_{b} that occurred after them, where $O^{\sigma}_{a}, O^{\sigma}_{b} \in R[a, b]$. For the precedence($a$, b) pattern, we do the opposite: we iterate through O^{σ}_{b} and count how many of these events have an event in O^{σ}_{a} that occurred before them. Finally, the succession and not-succession patterns emerge as byproducts of combining the results from both the response and precedence patterns. Specifically, the pattern succession(a, b) is valid only if the pair (a, b) appears in both the response and the precedence result sets, i.e. its calculated support exceeds the user-defined *support* threshold. Conversely, the not-succession(a, b) pattern is valid if the calculated support for both response(a, b) and precedence(a, b) is

less than (1-*support*) or if the *et-pairs* (a, b) and (b, a) do not exist in the IndexTable. The final support for succession(a,b) is computed as the product of the corresponding response and precedence pattern supports, while for the not-succession(a,b) pattern, the support is calculated as 2 − response(a,b) − precedence(a,b). Through slight modifications to how simple ordered relation patterns are extracted, we can detect both the alternate and the chain patterns as well.

Alternate Patterns. For the alternate response(a,b) the `count_occurrences` iterates through O_a^σ (O_b^σ for precedence) and count for how many of these events there is an event in O_b^σ (resp. O_a^σ) that appears after (resp. before) them, without any other event from O_a^σ (resp. O_b^σ) appearing in between.

Chain Patterns. For the chain response(a,b) pattern, the function will check for every event in O_a^σ (resp. O_b^σ for precedence) if there is an event in O_b^σ (resp. O_a^σ) that appears immediately after (resp. before) this one. However, this check requires additional information about the position of the events in the trace, which can be found in the SequenceTable, where the raw traces are stored. A different approach involves slightly modifying the index-building procedure of SIESTA and storing the position of the events instead of their timestamps in the IndexTable. Since the time information is not relevant in any of these Declare patterns, we opted for the second approach.

Once all occurrences of the pattern are counted, the final step is to assess which ones surpass the user-defined *support* threshold. Calculating support by simply dividing the number of traces containing the pattern by the total number of traces is not accurate. This is because multiple instances of the pattern can appear in a single trace, and two event types may appear infrequently but still together. Therefore, for ordered relation patterns, we calculate support as the number of pattern occurrences (or rule fulfilments) divided by the total appearances of the first constraint event in the log file (or the total appearances of the second event for the precedence pattern), also known as rule activations. The total occurrences of each event can be easily extracted with a linear scan of the SingleTable (line 7). Finally, the function *filter* (line 8) removes all the patterns that have a support less than the *support* threshold.

Example: Let us consider the extraction of the response constraints from the small logfile presented in Fig. 1. In lines 1–2 of Algorithm 1, we extract the ids of the traces that contain at least one instance of the pairs (A,B), (A,C), and (B,C), which are $[t_1, t_2]$, $[t_2]$, $[t_2]$, respectively. Next, in lines 3–5, we join the records from the SingleTable and maintain only the valid traces, resulting in the following output: R[A,B]: [(t_1,[1,2],[3]), (t_2,[1],[2])], R[A,C]: [(t_2,[1],[3])], and R[B,C]: [(t_2,[2],[3])].

In line 6, we count the occurrences of the response patterns and we find that response(A,B) occurred 3 times, while response(A,C) and response(B,C) occurred only once. Line 7 extracts the total occurrence of each event type from the SingleTable, which are 3, 2, and 1, respectively, for events A, B, and C. The corresponding supports are 1, 0.33, and 0.5, respectively. If we opted for chain-response patterns, we would have counted that chain-response(A,B) occurred 2 times, chain-response(A,C) did not occur, and chain-response(B,C) occurred only once. The corresponding supports are 0.66, 0, and 0.5, respectively. Assuming that a user would only be interested in the most frequent patterns and set the support threshold to 0.9, only the pattern response(A,B) would have been returned (line 8).

Existence and unordered relation patterns

All existence patterns can be extracted using three structures (Fig. 2):

I For each *et-pair*, this structure contains all the unique traces that contain this pair. It is straightforward to extract it from the IndexTable.

S For each event type x, this structure contains a list of tuples $(\sigma_{id}, O_x^{\sigma})$, where O_x^{σ} is the total occurrences of the event in the trace with id equal to $trace_{id}$ (Definition 5). Extracting S from the SingleTable is straightforward.

U This structure maps each event type to the total number of unique traces that contain it. It can be extracted from S.

Existence, Absence, and Exactly constraints. These three patterns can be answered solely by utilizing structure S. For existence patterns, we set n to the maximum observed occurrences of an event in a trace. In each iteration, we decrease n by one until it reaches zero, and we count how many traces have at least n occurrences of this event. Once the number of detected traces divided by the total number of traces in the log file exceeds the *support* threshold, we stop the iteration and add this pattern to the result set. A similar approach is taken for the absence pattern, but this time we set n to range from 1 to the maximum observed occurrences per trace. If the number of traces that have less than $n-1$ occurrences of the pattern divided by the total number of traces exceeds the *support* threshold, we stop and add this to the result set. Finally, for the exactly pattern, we evaluate all different values of n and count only the traces that have exactly n occurrences of the pattern.

Example: Executing the above processes with the *support* set to 1 will return the following patterns: existence(A,1), existence(B,1), and exactly(B,1), as both A and B appear in both traces at least once and B exactly once.

Co-existence, Choice, Exclusive Choice, and Responded Existence constraints. These patterns are based on how often two event types appear together, regardless of their order. For an *et-pair* (a,b), we can calculate the traces where these two events co-exist by utilizing the structure I. Specifically, by taking the union between the unique traces where the pair (a,b) occurs and the unique traces where (b,a) occurs (i.e., $I[a,b] \cup I[b,a]$), we can detect all the traces where events a and b co-exist. Furthermore, for every event pair (a,b), the following equation holds:

$$total\_traces = |I[a,b] \cup I[b,a]| + |OA| + |OB| + |N| \quad (1)$$

I
(A, A) -> [t1]
(A, B) -> [t1,t2]
(A, C) -> [t2]
(B, C) -> [t2]

S
(A) -> [(t1,2),(t2,1)]
(B) -> [(t1,1),(t2,1)]
(C) -> [(t2,1)]

U
(A) -> (2)
(B) -> (2)
(C) -> (1)

Fig. 2 Tables I, S, and U for the example log presented in Fig. 1b

where OA represents the traces where only event a appears, OB represents the traces where only event b appears, and N represents the traces where neither event a nor event b appears. To compute $|OA|$, we simply need to subtract the number of unique traces where both a and b appear together ($|OA| = U[a] - |I[a,b] \cup I[b,a]|$) from the number of unique traces where a appears. Based on Eq. (1), we can detect the remaining existence patterns for an *et-pair* (a,b).

Co-existence occurs either when a trace contains both a and b or if it contains no one of them, i.e., $occurrences = |N| + |I[a,b] \cup I[b,a]| = total\_traces - U[a] - U[b] + 2 \times |I[a,b] \cup I[b,a]|$

Choice occurs when a trace contains at least one of a or b, i.e., $occurrences = |OA| + |OB| + |I[a,b] \cup I[b,a]| = U[a] + U[b] - |I[a,b] \cup I[b,a]|$.

Exclusive Choice occurs when a trace contains one of the a or b, but not both, i.e., $occurrences = |OA| + |OB| = U[a] + U[b] - 2 \times |I[a,b] \cup I[b,a]|$

Responded Existence occurs either when a trace does not contain a or if it contains both a and b, i.e., $occurrences = total\_traces - |OA| = total\_traces - U[a] + |I[a,b] \cup I[b,a]|$.

Note that the difference from the co-existence is that we consider the traces that contain b but not a as valid traces.

Example: In our running example, executing the above processes with *support* set to 1, will return the following patterns:

- co-existence(A,B): A and B co-exist in both traces.
- choice(A,B): At least one of the A or B is present in both traces.
- choice(A,C): At least one of the A or C is present in both traces.
- choice(B,C): At least one of the B or C is present in both traces.
- responded-existence(A,B): Where there is an A, there is also a B in both traces.
- responded-existence(C,A): Where there is a C, there is also an A in both traces.
- responded-existence(C,B): Where there is a C, there is also a B in both traces.

Note that for co-existence, choice, and exclusive-choice, the order of event types is irrelevant. Therefore, in these rules, we consider only pairs where the event types follow a specific order (e.g., lexicographic). This ensures that no pairs are missed and that the mined set is the smallest possible set up to symmetry.

Position patterns

Position patterns are straightforward to extract from the SequenceTable. For each trace, we extract the first and last events and count their occurrences. If the calculated support is greater than the *support* threshold, the pattern is added to the result set. Note that, if the *support* threshold is set to a value higher than 0.5, at most one pattern of init and last will be returned. For instance, in our running example, extracting position patterns with a *support* threshold set to 1 will return only the pattern: init(A).

Incrementally mining declare constraints

In big data scenarios, we may want to extract constraints at runtime, while the log keeps increasing as new process instances run. In addition, in scenarios where data volume

increases significantly, extracting Declare constraints becomes challenging, even for scalable tools like SIESTA and processing the log in batches is preferred. Both scenarios can be supported by adopting an incremental extraction approach. Instead of waiting for all data to be indexed before initiating the mining process, we can couple constraint extraction with incremental indexing so that constraints can be efficiently extracted as new events are added to the system.

To this end, we introduce a new module to implement this logic on top of SIESTA (as shown in the updated architecture in Fig 3). This module called **Incremental Declare**, acts as an intermediate processing step following the index-building phase. It calculates key metrics (also referred to as *state*), which are sufficient to compute the support for all possible Declare constraints, and stores them in the Database Layer. Note that this approach is similar to MINERFul (Ciccio and Mecella 2015), which first creates indices to extract all possible Declare constraints, and then filters out those that do not fulfil the defined support threshold. In each execution, the module utilizes only the traces that changed in the last batch and the previously stored state.

We designed Incremental Declare in a manner that can be executed independently, although it is integrated with the main components of SIESTA, to preserve its general-purpose nature and offer flexibility. This allows the new module to be executed at varying frequencies. For example, a company may add new event logs every four hours, but it can choose to execute this module on a daily or weekly basis based on its needs.

Finally, we modified the Query Processor to extract Declare constraints directly from the stored state. As we will demonstrate in the experiments, the size of the state remains constant regardless of the number of indexed events and contains all the necessary information to compute the support for all possible constraints. Consequently, the response time for mining queries remains stable, irrespective of the log data volume consumed or user-defined support thresholds.

Algorithm 2 provides an overview of the incremental Declare mining approach. The algorithm takes as input the complete set of traces containing at least one new event, which can be retrieved from the SequenceTable based on the timestamp of the module's last execution. In line 1, the previously stored state is loaded from the database. This

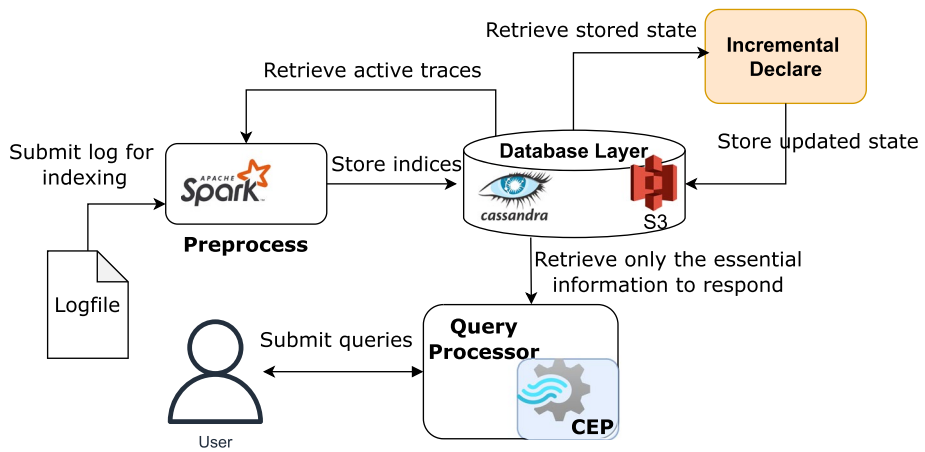


Fig. 3 SIESTA architecture including incremental declare

state consists of a collection of five tables that capture information on (i) event pairs that have not co-occurred in the indexed events, as well as metrics to calculate (ii) position constraints, (iii) existence constraints, (iv) unordered constraints, and (v) ordered constraints. Once the relevant information is extracted from the new batch of events (lines 2–6), it is merged with the previous state and stored back in the database (line 7).

Algorithm 2 Incremental declare mining

```
   Input modified_traces
1: state ← load_state_from_database()
2: neg ← find_non-existing_pairs(state,modified_traces)
3: p ← position_constraints(modified_traces)
4: e ← existence_constraints(modified_traces)
5: u ← unordered_constraints(modified_traces, neg)
6: o ← ordered_constraints(modified_traces, neg)
7: update_state(state,p,e,u,o,neg)
```

Let us take a closer look at the different procedures. The procedure in line 2 is responsible for detecting event pairs that do not occur in any trace. Specifically, it checks whether any non-existing pairs stored in the state appear in the new batch of events. Based on these pairs, constraints such as 'not succession' and 'not co-existence', which have 100% support, are extracted.

The `position_constraints` procedure calculates changes in position constraints. Specifically, for each trace in *modified_traces*, it extracts the activity of the last event. For each new trace (i.e., a trace that started in the current batch), it extracts the activity of the first event. Finally, for each trace that continued from a previous batch, it extracts the last activity recorded before the new batch of events. This information is then grouped, resulting in records formatted as ['first/last', *activity*, *occurrences*], where *occurrences* indicates the number of traces whose first (or last) activity matches the specified *activity*.

The `existence_constraints` procedure creates records in the format [*activity*, *n*, *occurrences*], where *occurrences* represents the number of traces that contain exactly *n* appearances of the activity *activity*. Using this information, it becomes straightforward to extract all Declare constraints in the existence category. The function in line 5 of the algorithm is responsible for maintaining an updated version of the structures U and I (as defined in Existence and unordered relation patterns section). As previously demonstrated, these structures contain all the necessary details to extract the complete set of unordered constraints.

Lastly, the procedure `ordered_constraints` handles the extraction of new ordered constraints. For each new event, it determines the different types of precedence rules that are activated and fulfilled and the different types of response rules that are fulfilled. This function creates records in the format [*rule*, $activity_a$, $activity_b$, *occurrences*], where *rule* can take the values: 'precedence', 'alternate precedence', 'chain precedence', 'response', 'alternate response', and 'chain response'. The field *occurrences* represents the number of times the corresponding rule is satisfied for activities $activity_a$ and $activity_b$. Note that it is not necessary to maintain separate information for the other ordered constraints, as they can be derived from these rules and the non-existing pairs extracted in line 2.

The records extracted from the various procedures for the new batch are merged with the previous state and stored in the database. The size of the state is independent of the

data volume and holds the minimal necessary information to support incremental mining and efficiently extract the complete set of constraints for any given threshold. Figure 4 presents the state after the execution of the Incremental Declare component for the example log presented in Fig. 1b.

A key difference between this approach and the one presented in the previous section is that we do not utilize SIESTA's indices. This is because we aim to calculate the support for all possible Declare constraints, leaving no opportunity for pruning, which is the primary purpose of an index. Furthermore, this incremental process is designed with the vision of an online Declare miner, incrementally extracting Declare constraints from a continuous stream of events, which we plan to implement in the future. In a near-real-time streaming environment, accessing the database for each new event is impractical; thus, the required data should always reside in the main memory. In the proposed solution, we demonstrate that it is possible to incrementally extract Declare constraints by utilizing only the events from the active (i.e., unfinished) traces.

Temporal variant analysis

Constraint mining in a general-purpose tool such as SIESTA can provide a multitude of ways of exploitation depending on the scope, one of which is to use past knowledge to make valuable inferences about the new arriving data. To support our case that a BDPA can efficiently support declarative process mining tasks, we present here SIESTA's capability to perform advanced variant analysis, a key process mining task (Dumas et al. 2018). For instance, consider a loan-application process where the event log consists of traces that end with either a "Loan Approved" or a "Loan Rejected" event. Using this criterion, we can split the event log into two distinct sublogs: one containing all traces that resulted in approved loans and another containing traces that ended in rejections. This segmentation enables a focused analysis of the events

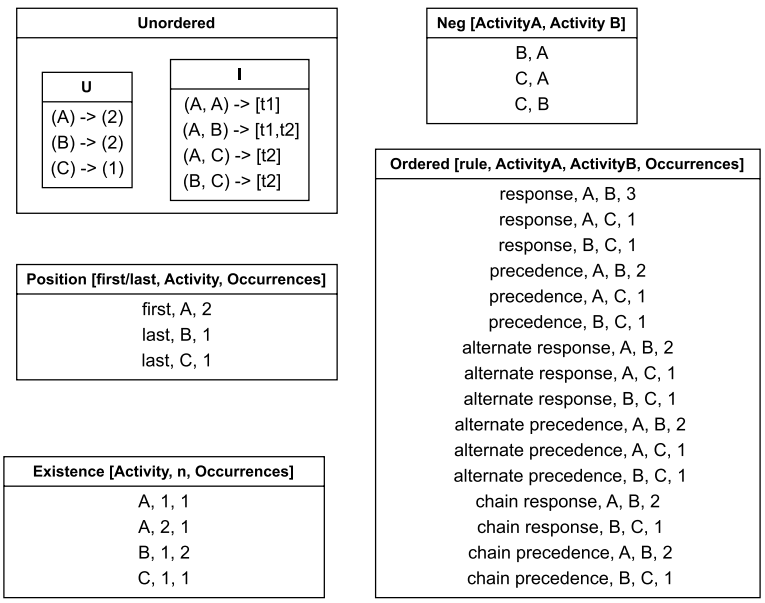


Fig. 4 State after the execution of the incremental declare component for the example log presented in Fig. 1b

leading up to each outcome. For example, traces in the "Loan Approved" log partition may display a consistent pattern of events that contributed to the positive outcome and does not appear in the other log partition, e.g., due to missing activities.

While the proposed mining approach in the previous section can provide all possible constraints we could use to split and analyze an event log, such as the *last* positional constraint utilized in the above example, this is not always adequate. In many real-world scenarios, it is equally important to detect not just compliance but also deviations over time. Temporal patterns serve as a powerful criterion for log splitting, as they capture the temporal dynamics and dependencies between activities in a process. Splitting based on these patterns allows for the identification of temporal deviations that may indicate problematic cases.

Imagine a scenario involving an insurance claim approval process, where traces either end with a "Claim Approved" or "Claim Rejected" event. Suppose that constraint mining reveals a temporal pattern between the activities "Document Review" and "Claim Assessment". Specifically, it finds that, in approved cases, the time between these two activities is typically less than 3 days. On the other hand, rejected claims show a different pattern, where the duration between "Document Review" and "Claim Assessment" often exceeds 5 days, indicating a temporal violation in the process. Using this type of temporal variant analysis, the event log can be split into partitions based on whether traces violate or satisfy this temporal pattern. That is, traces where "Document Review" and "Claim Assessment" are completed within the 3-day threshold can be grouped together as "Potentially Successful Claims". Conversely, traces where the duration exceeds 5 days are separated into another log partition representing "Potentially Problematic Claims". This split not only aids in post-hoc analysis to understand delays but also has predictive power. If an ongoing process instance exhibits a duration between these two activities approaching or exceeding the 5-day time span, it could be flagged as a high-risk case likely to be rejected. Consequently, exploiting temporal violations in this manner enables better control over process outcomes and the ability to predict and mitigate adverse scenarios before they fully materialize.

Temporal violations can cover a wide range of cases and are formally defined as follows:

Definition 6 (**Temporal violation**). Given an event log $\mathcal{L}$, a temporal violation occurs when a frequent constraint, defined as a recurring pattern of activities that conform to a specific rule in a trace like those in Table 2, exhibits a statistically significant deviation in its duration.

In our work, we focus on the following variant analysis-oriented case, as an example of the problems that can be addressed by our approach:

Definition 7 (**Temporal violation on log partitions**). Given two disjoint event log partitions L_1 and L_2 of an event log $\mathcal{L}$, a temporal violation occurs for a globally frequent constraint $C(a, b)$ if difference of the mean durations between occurrences of relevant activities in L_1 and L_2 exceeds a predefined threshold.

Note that L_1 and L_2 should always refer to the same business process, i.e., $L_1 \cup L_2 \subseteq \mathcal{L}$, and represent a log partitioning based on a specific criterion (e.g., positive and negative outcomes of the process instances contained in $\mathcal{L}$), ensuring that temporal divergence reflects systematic process variations rather than arbitrary differences.

In the following A naive approach to temporal violations extraction and Temporal violations extraction on top of SIESTA sections, a naive approach of detecting temporal patterns and mining violations as well as an optimized implementation of it on top of SIESTA are presented, respectively. We will focus on the **alternate response constraint** of the response pattern category (see Table 2), which is directly supported by SIESTA's indexing mechanism. As discussed in Mining declare constraints on top of SIESTA section, the proposed approach covers also all the additional Declare constraints.

A naive approach to temporal violations extraction

Detecting temporal patterns from the perspective of violations initially appears to be a computationally demanding problem. This complexity arises because this task inherently involves addressing three distinct computational challenges: (i) identifying all event pairs that do not overlap in time, exactly as in SIESTA's indexing phase (Mavroudopoulos and Gounaris 2024), (ii) recognizing frequent event pairs that are shared between logs, and (iii) statistically estimating their temporal deviation.

A naive approach to detecting temporal violations (TV) given a single log file is presented in Algorithm 3. It begins by taking an event log $\mathcal{L}$ and partitioning it into two sub-logs, based on a specified trace attribute, such as a specific resource value or a final event's attribute. It then identifies frequent event pairs in each subset that do not overlap in time, ensuring that only pairs meeting or exceeding a support threshold *sup* are retained. Next, the algorithm computes the intersection of these sets to find common frequent patterns. For each common pattern (e_i, e_j), it calculates the mean timestamp differences in both logs separately along with the corresponding standard deviation. The algorithm then checks for temporal violations by evaluating whether the absolute difference between the two mean times exceeds a threshold defined by a factor f multiplied by the standard deviation of the partition regarded as baseline (e.g., L_1; containing the positive-outcome executions of the process). In other words:

$$TV(a,b) = |\mu_{L_1}(a,b) - \mu_{L_2}(a,b)| > f \cdot stdev_{L_1}(a,b)$$

where (a,b) refers to *alternate response*(a,b), $\mu_{L_i}(a,b)$ is the mean duration between a and b in L_i, and $stdev_{L_1}(a,b)$ is the standard deviation of duration in L_1.

Algorithm 3 Naive extraction of temporal violations

Input event log $\mathcal{L}$, support threshold sup, sensitivity factor f, attribute $attr$
Output *Temporally violated patterns*
1: $L_1, L_2 \leftarrow \textbf{split}(\mathcal{L}, attr)$
2: $P_1 \leftarrow \{(e_i, e_j) \mid e_i \prec e_j \wedge \text{Support}_{L_1}[(e_i, e_j)] \geq \frac{sup}{|L_1|}\}$
3: $P_2 \leftarrow \{(e_i, e_j) \mid e_i \prec e_j \wedge \text{Support}_{L_2}[(e_i, e_j)] \geq \frac{sup}{|L_2|}\}$
4: $V \leftarrow \emptyset$
5: **for each** $(e_i, e_j) \in P_1 \cap P_2$ **do**
6: $\quad \mu_1^{e_i,e_j}, stdev_1^{e_i,e_j} \leftarrow \textbf{calculate\_statistics}((e_i, e_j), L_1)$
7: $\quad \mu_2^{e_i,e_j}, stdev_2^{e_i,e_j} \leftarrow \textbf{calculate\_statistics}((e_i, e_j), L_2)$
8: $\quad$ **if** $|\mu_1^{e_i,e_j} - \mu_2^{e_i,e_j}| > f \cdot stdev_1^{e_i,e_j}$ **then** $V \leftarrow V \cup \{(e_i, e_j)\}$
return V

This approach is considered naive primarily because it independently mines frequent patterns in each partition, followed by a comparison. This step-by-step process leads to increased computational costs as it requires repeatedly traversing the event log, first to partition and then to compute frequent patterns in both sub-logs. A key bottleneck also arises during the pattern identification phase, where each subset of events is analyzed separately, leading to redundant operations and memory consumption, especially when logs are large. Furthermore, calculating the mean and standard deviation of time differences for every common pattern instance and comparing these values can be computationally expensive, as it requires iterating over all pairs in both sub-logs. This results in multiple passes over the data to maintain statistical measures, which scales poorly with increasing log size and the number of detected patterns. Thus, the approach suffers from high computational overhead and does not provide efficient handling of temporal comparisons, making it unsuitable for large-scale event logs or real-time analysis scenarios. An optimized method is presented next.

Temporal violations extraction on top of SIESTA

In this section, we present an optimized implementation of the aforementioned algorithm for detecting significant temporal deviations in an event log. This constitutes an extension of the Query Processor's functionality, leveraging structures created during the log's preprocessing phase. Specifically, we examine the case where, given an indexed set of events from a single log file, we search within the contained traces for those event patterns that frequently occur in both parts of a non-random partitioning of the set, but exhibit significant temporal deviations between them. The set partitioning refers to splitting the initial event log in two parts, based on a specific trace characteristic. It is worth noting that the partitioning of the event log is performed on the set of processed data available at the time of the query, eliminating the need to maintain the set continuously partitioned according to a single exclusive attribute.

Algorithm 4 Detection of temporal violations in SIESTA

Input event log $\mathcal{L}$, support threshold sup, sensitivity factor f, attribute $attr$, k
Output *Temporally violated patterns*
1: $Pairs \leftarrow \emptyset$
2: **for each** $(e_i, e_j) \in \mathbf{IndexTable}_{\mathcal{L}}.keys$ **do**
3: $\quad Pairs \leftarrow Pairs \cup \{(e_i, e_j) \mapsto \mathbf{IndexTable}_{\mathcal{L}}[(e_i, e_j)]\}$
4: $Filtered \leftarrow \emptyset$
5: **for each** $(e_i, e_j) \in Pairs.keys$ **do**
6: $\quad Filtered \leftarrow Filtered \cup \{(e_i, e_j) \mid \left|Pairs[(e_i, e_j)]\right| \geq sup\}$
7: $Divergent \leftarrow Filtered.\mathbf{keep\_top\_divergent}(k)$
8: $D_1, D_2 \leftarrow \mathbf{split}(Divergent, attr)$
9: $V \leftarrow \emptyset$
10: **for each** $(e_i, e_j) \in D_1 \cap D_2$ **do**
11: $\quad \mu_1^{e_i,e_j}, stdev_1^{e_i,e_j} \leftarrow \mathbf{calculate\_statistics}((e_i, e_j), D_1)$
12: $\quad \mu_2^{e_i,e_j}, stdev_2^{e_i,e_j} \leftarrow \mathbf{calculate\_statistics}((e_i, e_j), D_2)$
13: $\quad$ **if** $\left|\mu_1^{e_i,e_j} - \mu_2^{e_i,e_j}\right| > f \cdot stdev_1^{e_i,e_j}$ **then** $V \leftarrow V \cup \{(e_i, e_j)\}$
return V

Our implementation, abstractly presented in Algorithm 4, utilizes the records stored in the IndexTable by the preprocessing component, in order to instantly get all the non-overlapping in time event pairs along with their instances' data (i.e., trace id and position occurrences). At this first step, all possible event pairs are fetched and accumulated in the map-type structure *Pairs*. In contrast to the naive approach, the filtering and preservation of frequent event pairs are performed prior to splitting the event set into two parts. It should be noted, however, that in order to maintain the high frequency of patterns in the resulting subsets-based on the attribute *attr*-we recognize that the support threshold *sup* must be set sufficiently high. This aligns with our objective of identifying significant (and therefore highly frequent) patterns with temporal divergences, that is we do not aim to differentiate log partitions according to the existence of patterns or not, but according to their different temporal properties while the same patterns exist in both partitions.

The splitting phase is optimized as well. First of all, we further narrow the pattern comparison space by retaining only the frequent patterns that exhibit the greatest variation in their temporal durations across different instances in $\mathcal{L}$. In other words, we retain the top-k most temporally-diverging patterns, noting that k is handled as a percentage of the total size of the *Filtered* event pairs. The *Divergent* structure contains these patterns along with the trace ids and positions their instances refer to. Thereafter, we split it in two partitions (D_1, D_2) by separating the event instances, based on their corresponding traces' characteristics (*attr*). According to the previous discussion regarding our rationale, $D_1 \cap D_2 \neq \emptyset$.

Consider the following *example*: Let $\langle a, b\rangle : \{(trace_{id} = 1, pos_a = 1, pos_b = 3), (trace_{id} = 2, pos_a = 5, pos_b = 6)\}$ be an element of *Divergent*. If trace with $id = 1$ was classified into D_1 and trace with $id = 2$ into D_2, then the two partitions hold two different instances of $\langle a, b\rangle$, i.e., $\langle a, b\rangle : \{(trace_{id} = 1, pos_a = 1, pos_b = 3)\} \in D_1$ and $\langle a, b\rangle : \{(trace_{id} = 2, pos_a = 5, pos_b = 6)\} \in D_2$.

Subsequently, for each pair $p = (e_i, e_j) \in D_1 \cap D_2$, we calculate its mean duration and standard deviation for both partitions, and, finally, we classify the event pairs that lasted with significant difference in average to the corresponding mean duration of D_1 as violating ones. In other words, the violating patterns are those whose average duration between the two partitions differs more than the standard deviation of their duration

in D_1, multiplied by a small factor f, which determines our sensitivity to the deviation thresholds that render the average duration as an outlier. A subtle notice is that we imply that D_1 is considered as the baseline event-instance set.

Evaluation

In our previous work (Mavroudopoulos et al. 2024), we have shown the superiority of the approach in Mining declare constraints on top of SIESTA section over Declare Miner and MINERful. Here, we conduct a whole new set of experiments, to provide further evidence regarding the behavior of our proposed solution and to evaluate the extensions of our approach described in Incrementally mining declare constraints and Temporal variant analysis sections. The research questions driving the evaluation are the following, so that they cover all the claimed contributions of this work:

- RQ1: How is SIESTA's indexing time affected by continuously arriving batches of events?
- RQ2: How is SIESTA's mining time affected by continuously arriving batches of events?
- RQ3: How does SIESTA scalability with regards to mining constraints compare to other state-of-the-art approaches?
- RQ4: What is the impact of the optimizations in mining temporal violation constraints with respect to the naive solution?

All the material in this section can be replicated through the codebase at https://github.com/siesta-tool/siesta-demo, where there is a dedicated docker file that can rerun all the experiments.

Setting

Datasets. Evaluating the performance and robustness of the proposed (incremental) approaches would require to use event logs larger than the ones used in our previous work (Mavroudopoulos et al. 2024), i.e., the *BPI Challenge 2017 - 2019* logs, but no such datasets are currently publicly available. To circumvent this, we consider one of the publicly event logs, namely the BPIC_2017 dataset, as our basis and we systematically extend it; however, in one of the key experiments we further employed the BPIC_2011 dataset, which contains much longer traces with a higher number of unique activities. The objective of this process is to simulate an event streaming environment, where batches of new and continuing traces arrive incrementally over time. This setup allows us to mimic the behavior of a data-intensive, real-world business process, where events are continuously and frequently generated, with new instances of the process initiated while existing ones evolve. A summary presentation of the characteristics of all evaluated datasets is shown in Table 3.

The synthetic dataset generation process is implemented based on a simple method of reusing event traces from a single log file. Initially, the original log is transformed to fit within the context of a single day, ensuring that the event timestamps are adjusted appropriately for each batch while maintaining the integrity of the original event sequence. Subsequently, traces from the original log are replicated for each batch, and

Table 3 Descriptive statistics of the evaluated original datasets

Log name	Total traces	Total events	Event classes	Trace length		
				min	avg	max
BPIC 2011	1,143	150,291	624	1	131	1,814
BPIC 2017	31,509	1,202,267	26	10	38	180

Table 4 Characteristics of the synthetic batch templates based on BPIC_2017

Batch type	Average traces	Average events	Unique traces[a]	Unique events[b]
All new	31,509	1,202,267	100%	2%
Replicas	31,509	1,202,267	100%	0%
All continue	17,202	201,574	0%	11%
Partial continue	28,461	905,030	90%	3%

[a] Refers to the percentage of globally new trace ids appeared in a batch file (with the first batch excluded)

[b] Refers to the percentage of globally new event types appeared in a batch file (with the first batch excluded)

new case identifiers are assigned. This method guarantees that each batch contains a distinct set of traces, simulating various instances of the business process executed in a random order. To generate a more realistic scenario of trace continuation, a portion of the original traces is randomly selected to continue in subsequent batches across the generated dataset. Overall, the following alternative approaches were tested regarding the replication and reproduction of the events and traces (see also Table 4):

- **All traces are new.** Each batch file contains only new traces characterized by new case IDs and new event sequences, i.e., every batch exhibits different event patterns.
- **Same traces as new (Replicas).** Each batch file contains exactly the same distinct event sequences with a different case ID, i.e., the same distinct traces with different IDs in each batch.
- **All traces continue.** Each batch file contains only trace sequences that started in the first batch and ended in the last one.
- **Some traces continue (Partial).** Each batch contains the continuation of a predefined portion (in our case we used 10%) of previous batches' traces, while the rest are previous complete traces as new, as described above.

Figure 5 shows the results obtained simulating these methods. We can see that in all cases, except the one where all cases keep evolving, the indexing time per batch remains stable. Thus, in the remainder of the evaluation, we use only the Partial method, because it is more realistic and suitable for evaluating the system's performance and scalability. In practice, using the Partial method, we can ingest over 1M events in less than 2 minutes using 1 commodity machine (details are provided below).

The goal of this preliminary experiment is to assess how well the system can handle increasing loads of event data as the number of streaming log partitions (batches)

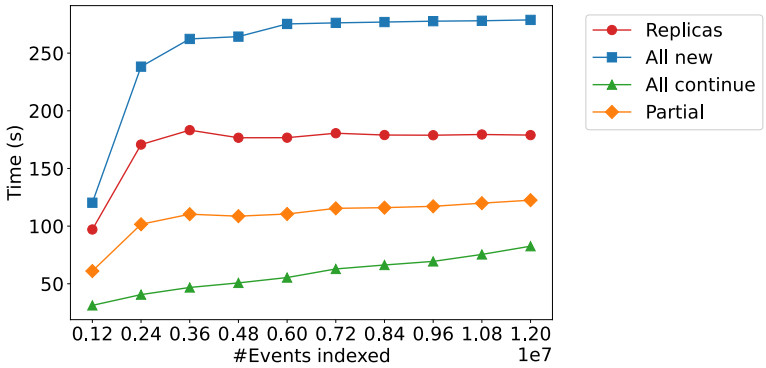


Fig. 5 SIESTA's indexing time per batch while overall events' volume increases; the batch size is approx 1.2M events, which is the size of the original BPIC_2017 dataset

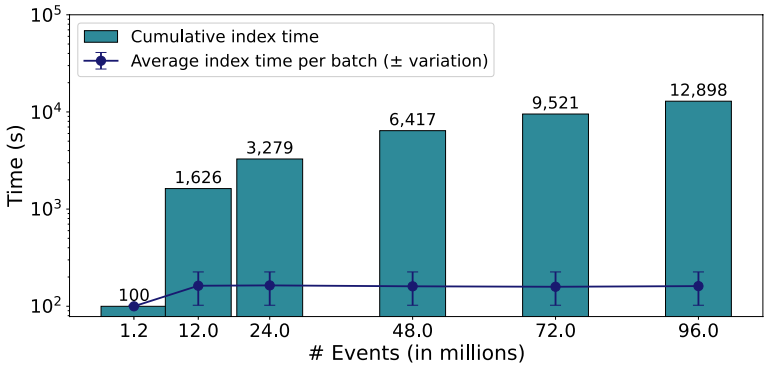
rises even under extreme configurations, and how partial compares against such configurations. Although Fig. 5 reports the results with only 10 batches, we created datasets with up to 80 batches, obtaining the same behaviour.

Competitors. We evaluate the proposed incremental mining approach in comparison to our previous work on SIESTA's Query Processor (QP Declare), focusing on scalability and temporal performance. Additionally, we compare our new approach against the two best-performing competitors from our previous evaluation, MINERful and the Declare Miner implemented in RuM (version 0.7.2). Since the proposed methods do not support constraint pruning based on hierarchies or other criteria, all methods were configured to report the complete set of constraints without any filtering. Furthermore, the technical extension to SIESTA for mining temporal violations is evaluated against the baseline naive algorithm, as described in Temporal variant analysis section.

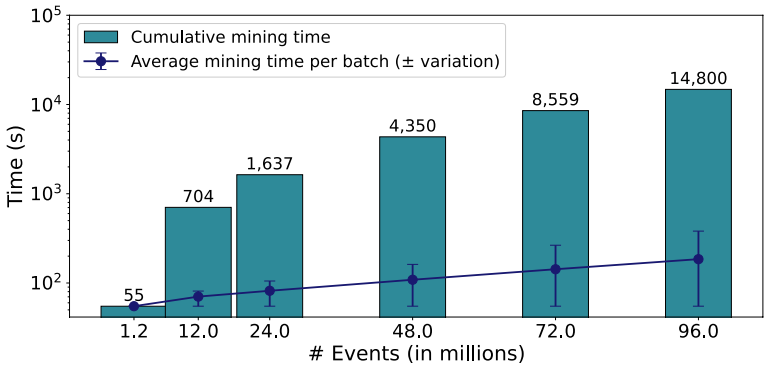
Infrastructure. Experiments presented in Mining declare constraints and Mining temporal violation constraints sections were conducted on a single machine running Ubuntu 20.04, equipped with 64GB of RAM and an 8-core (16-thread) CPU clocked at 2.10GHz. Each approach was allocated 50GB of RAM and granted full access to the CPU resources, allowing efficient utilization through parallelization (required, for instance, by RuM, which spawns multiple threads). In addition to the modifications made to the structure of the S3 tables (as described in Preliminaries section), we further improved the efficiency of SIESTA's incremental indexing by removing completed traces from the LastChecked tables at the end of the indexing process. This step does not introduce any errors, as no new events are appended to these traces once they are completed.

Mining declare constraints

We begin by answering RQ1, assessing the performance of SIESTA's incremental indexing after implementing the optimizations, using the extended BPIC_2017 dataset. The results are shown in Fig. 6a. This figure presents the cumulative indexing time, i.e., the total time spent for all batches, and the average preprocessing time per batch. Notably, after the initial execution, during which no previous data is available, the average indexing time remains constant for the same volume of indexed events, which was the intended outcome.



(a)



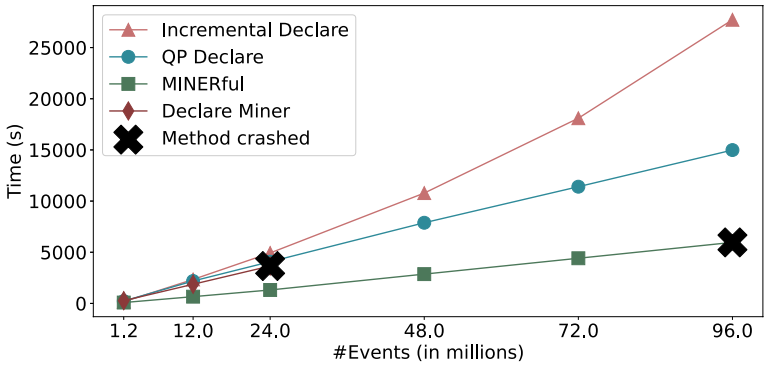
(b)

Fig. 6 Performance of **a** SIESTA's incremental indexing and **b** Incremental Declare as the data volume increases

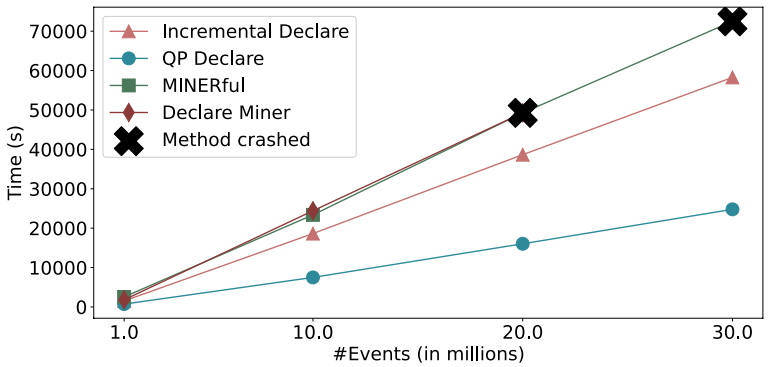
Next, we shift our focus to RQ2. Figure 6b illustrates the cumulative mining time and the average mining time per batch of Incremental Declare. Although the mining time increases, it remains relatively low given the data volume. The consistent increase in average time per batch is primarily due to the growing number of events in the SequenceTable and SingleTable. Specifically, the method needs to read these indices and retain only the relevant events for the new batch.[5]

Regarding RQ3, Fig. 7 shows the time required by all approaches to extract the complete set of Declare constraints with over 90% support from the BPIC_2017 and BPIC_2011 datasets. Note that the BPIC_2011 was added to provide evidence that the results presented are not specific to a single dataset. For both methods implemented on top of SIESTA, namely Incremental Declare and QP Declare (described in Mining declare constraints on top of SIESTA section), we present the cumulative time required for indexing and post-processing the data (in the case of Incremental Declare) as well as the query execution time. SIESTA-based methods exhibit the highest response times due to the extensive index building. As discussed in our previous work (Mavroudopoulos

[5] Adapting this method to a streaming environment, where only incomplete traces are kept in main memory, is expected to yield a constant processing time per event, thus enabling real-time mining of Declare constraints.



(a) BPIC_2017



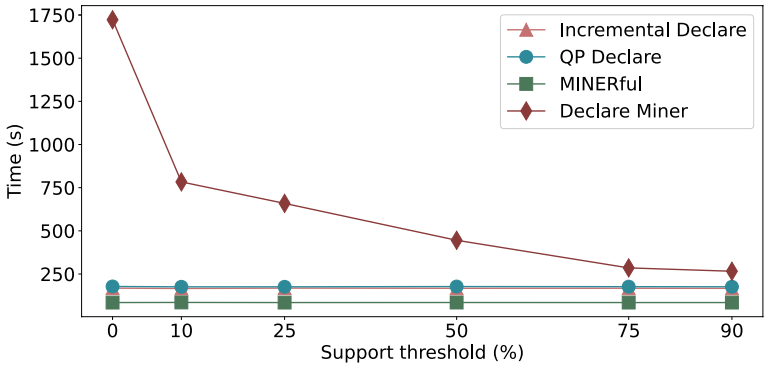
(b) BPIC_2011

Fig. 7 Evaluation of different approaches for mining the complete set of Declare constraints with 90% support. For the proposed methods, we present both the cumulative time required for indexing and post-processing the data (in the case of Incremental Declare) as well as the query execution time

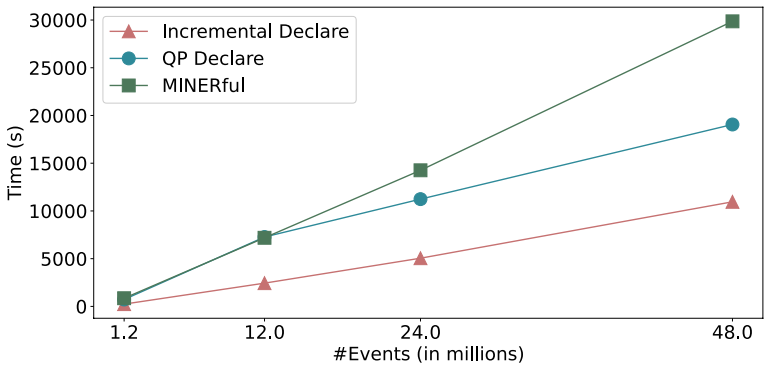
et al. 2024), the constructed indices are not fully utilized if the mining process is executed only once.

Note that SIESTA-based methods were the only ones capable of extracting Declare constraints from 96 and 30 million events of the BPIC_2017 and BPIC_2011 datasets, respectively. More specifically, Declare Miner could not complete the process after 24 million events, while MINERful ran out of memory for the BPIC_2017 dataset. As for BPIC_2011, they both failed to complete processing 20 and 30 million events, respectively. Although Incremental Declare demonstrated the worst performance in scenarios with a medium amount of unique events, it is important to note that since incremental mining took place after every batch, a user would have been able to extract all Declare constraints at any point in time for any given threshold in under 20 seconds, even for the 96 million events. This is because the query only requires information from the stored state and some aggregated values from the SingleTable, which are efficiently extracted due to effective partitioning.

Figure 7 presents the performance of the different methods for a single support threshold (90%). However, it is also important to evaluate how the support threshold affects the performance of these methods. We assessed the time required by each method to



(a) Robustness



(b) Multiple mining queries

Fig. 8 **a** Evaluation of the impact of the support threshold on the performance of various methods. **b** The cumulative time of the most scalable methods for executing the mining process 10 times (for 10 different support thresholds ranging from 0.8 to 0.89)

extract Declare constraints from the original BPIC_2017 dataset (1.2 million events) across various support thresholds, and the results are shown in Fig. 8a. As illustrated, the performance of both SIESTA-based methods and MINERful is independent of the support threshold. This is because all three approaches build indices capable of extracting all Declare constraints, even those with zero support, and subsequently filter the constraints based on the specified support level during the final step. In contrast, Declare Miner's performance is significantly impacted by the support threshold, as it uses this parameter to perform early pruning of constraints. Based on the results in Figs. 7 and 8a, it is evident that Declare Miner is not suitable for large data volumes and was therefore excluded from the subsequent experiments.

Finally, we present the total time required by each method to mine Declare constraints for 10 different support thresholds (ranging from 0.8 to 0.89) as the data volume increases. For this evaluation, we used the BPIC_2017 dataset. The results are shown in Fig. 8b. While Incremental Declare initially exhibited the worst performance when executing the mining process only once, its extensive preprocessing is compensated by the speed-up gains in the response times, as the number of queries increases (e.g., for different support thresholds or sets of constraints). The response time for one query was less

than 20 seconds, even with 96 million events indexed. Additionally, QP Declare demonstrated better performance than MINERful, as the built indices are increasingly utilized with more queries. For example, incremental declare is 2.23 times faster than QP declare and 2.82 times faster than MINERFul for 24M overall events.

Based on the findings exposed this section, we can draw the following conclusions:

- Declare Miner is not suitable for scenarios involving large data volumes, as it does not scale well and its performance is significantly impacted by the support threshold value.
- MINERful is the best option when the data and indices can fit into the main memory and the extraction process must be performed only once. Thanks to in-memory indices, MINERful can quickly discover the complete set of Declare constraints, regardless of the defined threshold.
- SIESTA demonstrated efficient support for incremental indexing. In real-world big data applications, where data are not likely to fit into the main memory, SIESTA-based methods emerge as the only viable solution. When mining of the constraints is needed frequently, it is advisable to spend additional time during preprocessing to prepare the data structure for efficient mining at any time, i.e, using the Incremental Declare SIESTA-based approach. However, if constraint mining is infrequent, QP Declare is the better option, being multiple times faster than the competitors while achieving a throughput of over 3 K events per second.

Mining temporal violation constraints

The following section evaluates the performance and quality of the optimized method for mining temporal violation (MTV) constraints, comparing it against the naive algorithm to answer RQ4. The experiments discussed below were performed on the BPIC_2017 original dataset, mentioned in Setting section. This dataset was selected for evaluation purposes, though other datasets from that section were also tested. The results remained consistent across different datasets, following the same overall trend as expected.

Figure 9 presents the impact of optimizations applied to the SIESTA MTV algorithm, which include early pair pruning and utilization of the IndexTable, allowing for

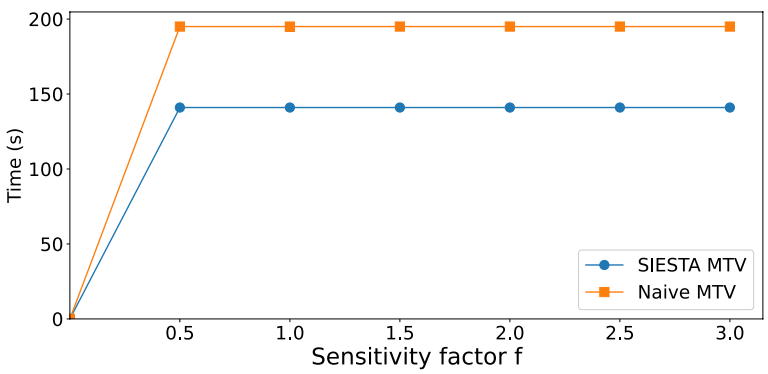
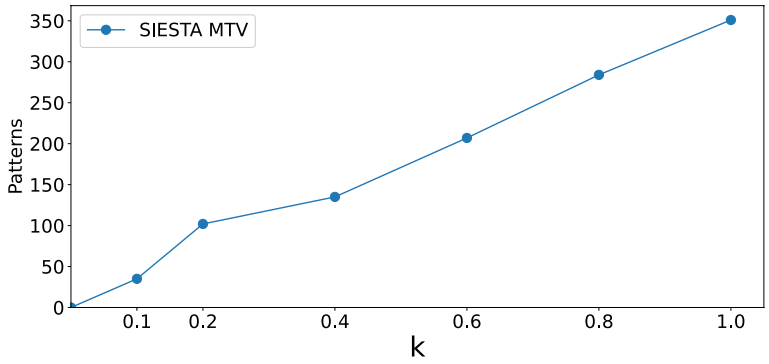
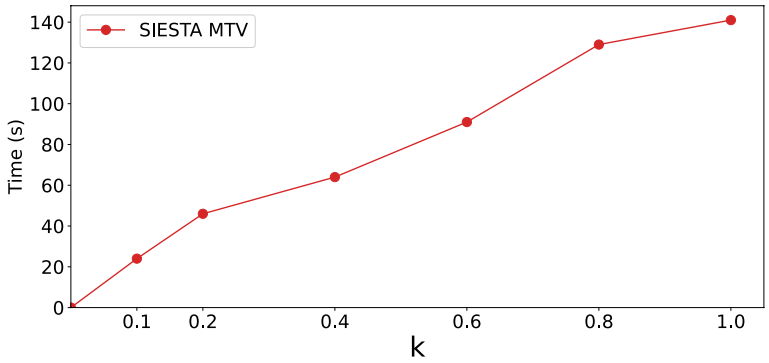


Fig. 9 Evaluation of the impact of the optimizations applied to the naive algorithm for MTV, for support threshold $sup = 100\%$ and $k = 100\%$



(a) Patterns



(b) Time

Fig. 10 Evaluation of the impact of different k values in **a** time and **b** number of patterns

pre-calculated event pairs -referring to potential alternate response constraints. This optimization leads to robust time performance across varying sensitivity factors. It should be noted that both SIESTA's and the naive method show deterministic behaviour, as expected, exhibiting constant time for mining; 141 and 195 seconds, respectively, showing roughly 1 minute difference. Despite not providing direct access to timestamps, the use of the SequenceTable maintains good time efficiency. This demonstrates that our approach can effectively manage time performance, even with restricted access to individual timestamps,[6] ensuring no significant delays in processing compared to the naive MTV approach, which suffers from performance issues due to its lack of efficient event pair discovery and filtering.

Additionally, by retaining only the top-k most divergent event pairs, we can control the number of patterns, keeping those showing the largest temporal divergence overall. This approach allows us to target the most significant violations, while maintaining control over the number of patterns processed. For increasing value of k, Fig. 10 shows a linear progression on both the number of patterns (Fig. 10a) and time (Fig. 10b). This behaviour is entirely expected in both cases. On the one hand, the linear increase in

[6] The structures D_1 and D_2, derived from the segmentation and filtering of the IndexTable, do not retain the complete information regarding the pattern instances. Instead, they store only references to the traces where these patterns occur and their respective positions within those traces. The corresponding timestamps are stored in the SequenceTable.

detected patterns is directly tied to the value of k, which represents a percentage of the total candidate patterns. On the other hand, the more interesting observation is the linear increase in processing time. We see that for a tenfold increase in k, the time increases by roughly 485%, which can be attributed to the additional computations of $\mu_1 - \mu_2$ differences. Applying this approach in an event stream environment would surely require further optimization.

Overall, the above evaluation of time and number of patterns between SIESTA and the naive approaches, proves a substantial improvement over the naive approach in performance for mining temporal violations in event logs. By leveraging optimizations such as early pair pruning and indexed event pairs, SIESTA maintains efficient time performance while significantly increasing the detection of meaningful violating patterns, allowing users to adjust the percentage of the most significant temporal patterns that interest them, and consequently benefit from the corresponding time trade-off.

Threats to validity

The validity of the proposed solutions and their evaluation are subject to several threats. Regarding internal validity, a general threat affecting this type of research is that we may have missed important configuration parameter values that could have improved the performance of the competitors. This threat has been mitigated by thoroughly reviewing the existing code bases of competitors, even consulting with their developers to seek clarifications when needed. Additionally, the large logs have been simulated by extending existing ones. Although, we have tested four different extension strategies to simulate different variability patterns, first-hand real-life large logs may reveal other challenges when applying the proposed approach that we did not anticipate in this work.

The generalizability of the results discussed in the paper is threatened by several factors. First, we rely on a widely adopted technique for expressing declarative process constraints, i.e., Declare, which allows us to compare our approach with existing ones. There are however other declarative languages that may be considered. Second, the proposed extension for temporal variant analysis relies on our own definition of temporal violations, which is not used by other approaches in the literature. While we hope that our definition may become a reference for future research, for the moment the evaluation of the experiments on temporal constraint violations can only rely on our own naive approach as a competitor. Finally, we consider a limited number and types of event logs considered, particularly as far as *large* event logs are concerned. This is yet another typical threat affecting this type of experimental research. To mitigate this threat, we have used real world, publicly available event logs that are widely used in the literature and that encompass different trace variability regarding both the distributions of case durations and number of events.

Conclusion and future work

This work provides strong evidence regarding the suitability and efficiency of existing general-purpose big data analytics frameworks in declarative process mining, thus alleviating the need to develop ad-hoc solutions. Specifically, we have introduced a scalable and efficient solution for extracting declarative process constraints from large event log files, by leveraging the recently proposed SIESTA system. By providing an extensive set

of experiments on event logs larger than the ones considered in our previous work, we have shown that our solution outperforms other state-of-the-art methods when dealing with very large event logs.

Besides catering for improved performance and scalability, general-purpose big data analytics frameworks can be flexibly extended to accommodate different data processing scenarios. In this work, we have shown how our solution can operate in an incremental manner, supporting constraint mining when event logs are continuously delivered in batches. The evaluation results show that our solution achieves a throughput of several thousand events per second using a single commodity machine by creating and maintaining efficiently a set of indexes across the batches of events. Additionally, we have also shown how the same framework can perform other declarative mining tasks, such as differentiating log partitions according to the temporal properties of the same patterns in each partition.

In the future, we aim to extend our implementation by introducing additional components, e.g., for conformance checking and anomaly detection over the indexed traces based on a list of rules. This will complement MTV and further enable the identification of outlying traces and changes in process execution. Furthermore, as outlined by Elgammal et al. (2016), beyond the discussed patterns, there exist two additional categories known as Timed and Resource Patterns. Integrating these patterns, as well as considering data payloads conditions activating the patterns, would be interesting for future work. We will also investigate how the proposed framework could be exploited in the context of probabilistic Declare constraint mining (Alman et al. 2022), which requires the mining of constraints with different support thresholds to discern the probabilities associated with the constraint activation and target conditions. Finally, in our broader plans, predictive process monitoring over the SIESTA infrastructure is included.

Authors' contributions
A.G. and M.C. supervised the project, framework development and paper writing. I.M. developed the framework and executed the experiment, together with C.B., K.V., and G.K.

Funding
This work was supported by the NRF Korea, Grant Number 2022R1 F1 A1072843.

Data availability
No datasets were generated or analysed during the current study.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 28 October 2024 Accepted: 21 April 2025
Published: 25 April 2025

References

Alman A, Ciccio CD, Haas D, et al (2020) Rule mining with rum. In: Proc. ICPM. IEEE, New York, pp 121–128
Alman A, Maggi FM, Montali M et al (2022) Probabilistic declarative process mining. Inf Syst 109:102033

Augusto, A., Awad, A., & Dumas, M. (2021). Efficient checking of temporal compliance rules over business process event logs. arXiv preprint arXiv:2112.04623.
Back CO, Slaats T, Hildebrandt TT et al (2022) Discover: accurate and efficient discovery of declarative process models. Int J Softw Tools Technol Transfer 24(4):563–587
Burattin A, Cimitile M, Maggi FM et al (2015) Online discovery of declarative process models from event streams. IEEE Trans Serv Comput 8(6):833–846
Ciccio CD, Mecella M (2015) On the discovery of declarative control flows for artful processes. ACM Trans Manage Inf Syst 5(4). https://doi.org/10.1145/2629447
De Smedt J, Deeva G, De Weerdt J (2019) Mining behavioral sequence constraints for classification. IEEE TKDE 32(6):1130–1142
Di Ciccio C, Mecella M (2015) On the discovery of declarative control flows for artful processes. ACM Trans Manag Inf Syst 5(4):24:1-24:37
Donadello I et al (2022) Declare4Py: A python library for declarative process mining. In: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2022. Spinger: Cham. pp 117–121
Dumas M, Rosa ML, Mendling J et al (2018) Fundamentals of Business Process Management, 2nd edn. Springer, Cham. https://doi.org/10.1007/978-3-662-56509-4
Elgammal A, Turetken O, van den Heuvel WJ et al (2016) Formalizing and appling compliance patterns for business process compliance. Softw Syst Model 15:119–146
Esser S, Fahland D (2021) Multi-dimensional event data in graph databases. J Data Semant 10(1–2):109–141
Leno V, Dumas M, Maggi FM (2018) Correlating activation and target conditions in data-aware declarative process discovery. In: Business Process Management: 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9–14, 2018, Proceedings 16. Spinger: Cham. pp 176–193
Maggi FM, Bose RJC, van der Aalst WM (2012) Efficient discovery of understandable declarative process models from event logs. In: CAiSE. Spinger: Cham. pp 270–285
Maggi FM, Di Ciccio C, Di Francescomarino C et al (2018) Parallel algorithms for the automated discovery of declarative process models. Inf Syst 74(P2):136–152
Maggi FM, Dumas M, García-Bañuelos L et al (2013) Discovering data-aware declarative process models from event logs. In: Business Process Management: 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings. Spinger: Cham. pp 81–96
Maggi FM, Mooij AJ, van der Aalst WM (2011) User-guided discovery of declarative process models. In: IEEE CIDM. IEEE, Spinger: Cham. pp 192–199
Mavroudopoulos, I., & Gounaris, A. (2022). SIESTA: A scalable infrastructure of sequential pattern analysis. IEEE Transactions on Big Data, 9(3), 975-990.
Mavroudopoulos I, Gounaris A (2024) A comprehensive scalable framework for cloud-native pattern detection with enhanced expressiveness. CoRR abs/2401.09960. https://doi.org/10.48550/ARXIV.2401.09960
Mavroudopoulos I, Varvoutas K, Kougka G et al (2024) Exploiting general purpose big-data frameworks in process mining: The case of declarative process discovery. In: International Conference on Business Process Management. Spinger: Cham. pp 185–202
Navarin N, Cambiaso M, Burattin A et al (2020) Towards online discovery of data-aware declarative process models from event streams. In: 2020 International Joint Conference on Neural Networks (IJCNN). IEEE, Spinger: Cham. pp 1–8
Pesic M, Schonenberg H, van der Aalst WM (2007) Declare: Full support for loosely-structured processes. In: Proc. EDOC. Spinger: Cham. p 287
Schönig S, Solti A, Cabanillas C et al (2016) Efficient and customisable declarative process mining with SQL. In: Proc. CAiSE. Spinger: Cham. pp 290–305
van Beest N, Groefsema H, García-Bañuelos L et al (2019) Variability in business processes: Automatically obtaining a generic specification. Inf Syst 80:36–55
van der Aalst W, Carmona J (2022) Process Mining Handbook. Springer, Cham
van Dongen BF, de Medeiros AKA, Verbeek HMW et al (2005) The prom framework: A new era in process mining tool support. In: Ciardo G, Darondeau P (eds) Applications and Theory of Petri Nets 2005. Spinger: Cham. pp 444–454
Westergaard M, Stahl C (2013) Leveraging super-scalarity and parallelism to provide fast declare mining without restrictions. In: BPM Demos. Spinger: Cham. pp 31–35
Zaki NM, Helal IM, Hassanein EE et al (2022) Efficient checking of timed ordered anti-patterns over graph-encoded event logs. In: International Conference on Model and Data Engineering. Spinger: Cham. pp 147–161
Ziehn A, Grulich PM, Zeuch S, Markl V (2024) Bridging the gap: Complex event processing on stream processing systems. In: Tanca L, Luo Q, Polese G et al (eds) Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28. OpenProceedings.org, pp 447–460. https://doi.org/10.48786/EDBT.2024.39

Publisher's Note