Exploiting general purpose big-data frameworks in process mining: the case of declarative process discovery

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

[1] Aristotle University of Thessaloniki, Greece
{mavroudo,kmvarvou,georkoug,gounaria}@csd.auth.gr
[2] Ulsan National Institute of Science and Technology, South Korea
mcomuzzi@unist.ac.kr

Abstract. Declarative process models are a flexible tool to capture the process model in unstructured and highly variable business scenarios. Extracting declarative process constraints, however, is a computationally intensive task and, as the volume of log data increases, efficiently extracting constraint template occurrences becomes challenging. While there have been efforts to improve the performance of declarative constraint extraction, we argue that solutions can be seamlessly derived by adapting application-agnostic pattern analysis techniques for big data to this task. In this work, we propose a solution for the efficient extraction of declarative constraints, specifically those described in the Declare language, without the limitation of developing tailored systems for declarative processes. We build on top of a recent scalable framework, named SIESTA, which can perform efficient pattern analysis on large log files. Our approach yields promising results, significantly outperforming the existing Declare Miner and MINERful solutions.

Keywords: Business Process Constraints · Process Mining · Big Data.

1 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 [10]. 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 [1] 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 [13] are often more appropriate. These take the form of a set of constraints 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 [13]. Declare [19] is one of the languages that can be used to express such constraints. In process discovery and other scenarios, such as anomaly detection, calculating the support of constraints in a log can be crucial. In such cases, extracting Declare constraints can be a computationally challenging task. Intuitively, discovered Declare constraint templates must be checked against every trace in a log to verify their support. While some methods have been proposed to efficiently extract Declare constraints as logs grow larger by exploiting parallelism [3,16], we argue that more can be done by adapting general-purpose big data pattern analysis frameworks to the task of declarative process constraint extraction.

One such example of general-purpose big data pattern analysis framework recently published is SIESTA [18,17], 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. The primary focus of this work is the efficient extraction of all the patterns presented in the Declare modeling language [14,6] extending SIESTA's query processor and leveraging the built indices, thus alleviating the need to build tailored declarative pattern extraction tools in the BPM community. Declare patterns include the patterns usually employed in process mining, along with numerous additional ones, providing a more comprehensive collection of constraints.

The results of our approach are promising. We efficiently extracted the complete set of Declare patterns from three real-world, large business process log files, encompassing 1.2 to 2.5 million events, in less than three minutes using a commodity machine. In contrast, leading competitors such as the Declare Miner [15], used by RuM [2], required significantly more time. 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 our indices are better utilized. In such scenarios, Declare Miner required up to 15.6 times longer, while MINERful [7] was overall 1.63 times slower. Additionally, through artificially generated event logs containing hundreds of event types and up to 10M events, we demonstrate that our solution scales better than the competitors. Our implementation is publicly available.[3]Overall, our paper shows how the BPM community can leverage effectively existing general-purpose big-data analysis frameworks, without needing to rely on ad-hoc solutions specifically tailored to event log data.

The remainder is structured as follows: Sec. 2 and 3 present the related work and the background, respectively. Our contribution, showcasing how we leverage SIESTA's indices for efficient extraction of the Declare patterns, is presented in Sec. 4. Sec. 5 reports on the performance evaluation, while we conclude in Sec. 6.

[3] `https://anonymous.4open.science/r/SequenceDetectionQueryExecutor-046B`

2 Related Work

Our work relates to several other proposals in declarative process modelling. Extracting constraints from real-world processes by focusing on the discovery of Declare models in Linear Temporal Logic (LTL) based on event logs has been investigated by [15]. The main proposal comprises a two-phase approach employing the Apriori algorithm to generate a list of candidate Declare constraints. These are further pruned based on relevance using 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. Another approach that extends the rationale of [15] for parallel discovery of declarative process constraints is introduced by [16] and aims to improve the time performance exploiting concurrency. The work in [16] discusses a Declare Miner plug-in of the ProM PM toolkit [9] that uses Declare modeling for efficient model discovery by combining it with a group of algorithms for sequence analysis apart from Apriori. Additionally, the parallelization is achieved by two different partitioning methods: search space and database partitioning. Note that SIESTA is scalable by design and capable of benefiting from massive parallelism through its implementation in Spark and the underlying data storage layer.

The MINERful algorithm is another process mining approach proposed by [7] for discovering declarative process constraints. This is also a two-phase technique, where the first phase produces statistical information extracted from event logs, while the second one estimates the constraints. This approach has also been extended by [21], who implemented a tool, named UnconstrainedMiner, for fast and accurate mining 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, as shown by [7], MINERful is faster than UnconstrainedMiner. Both Declare Miner and MINERful have been implemented in the rule mining toolkit RuM [2]. RuM is a representative example of an artifact that integrates multiple Declare-based process mining methods. In terms of implementation, the Declare4Py package [8] has also been proposed. It implements several declarative process modelling techniques, but without an explicit focus on performance, particularly as far as declarative constraints discovery is concerned.

There are also other approaches, such as [5] and DisCoveR [4], that also discover and utilize declarative constraints similar to those described in the Declare language. Our solution efficiently discovers frequent patterns that adhere to Declare constraints, which can later be utilized in various applications. Unlike the approaches in [5] [4], 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, [5] 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,

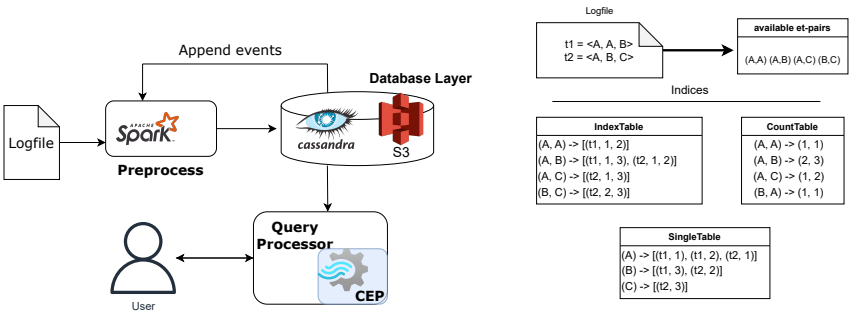
modeled as Dynamic Condition Response (DCR) graphs, that hold true in all traces of an event log. Therefore, these approaches normally yield a smaller set of patterns compared to our solution. Regarding expressivity, the method in [5] does not consider most of the unordered, existence, and position templates, as well as 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) [11] 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.

Finally, it is important to note that there is another set of open-source implementations [20,22,3,12] that perform process mining and pattern extraction from event logs using SQL or graph-based database architectures and technologies. The approach by [20] applies SQL queries to relational event data, while [12] 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 [22], 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 [3], where the recent MATCH_RECOGNIZE SQL extension is shown to be the dominant solution. In this paper, we also compare our proposed solution against both Neo4j and MATCH_RECOGNIZE.

3 Preliminaries

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 Sec. 4). SIESTA is an infrastructure for efficient support of sequential pattern queries based on inverted indices, generated from a logfile 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 L is defined as $L = (E, C, \gamma, \delta, \prec)$, where E is the finite set of events, C is the finite set of Cases, $\gamma : E \to C$ is a function assigning events to Cases, $\delta : E \to A$ 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 events' timestamp.*



(a) Architecture (b) Indexing example

Fig. 1: System overview

The event timestamps provide an ordering between the events that belong in a trace. 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 the main definitions.

Definition 2 (*et-pair*). *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)*, if* $x < y \wedge ev_x.type = a_i \wedge ev_y.type = a_j$ *and* $ev_x, ev_y \in \sigma$*. Note that based on this definition, the events in a 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 [17], *event-pairs* are generated based on Definition 4. However, in recent work [18], 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 $ev_1.type = ev_2.type = a_i$ (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 L, which is utilized for efficiently retrieving relevant events when calculating Declare constraints, as discussed in the next section.

SIESTA's architecture (illustrated in Figure 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 our implementation, we primarily utilize Apache Cassandra[4], but we also evaluate S3 as an alternative storage solution for big data scenarios. Finally, the query processor utilizes the stored indices to perform efficient pattern analysis and integrates a Complex Event Processing (CEP) engine.

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 $trace_{id}$ 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 SequenceTable contains the raw traces, while the CountTable holds basic statistics for every available *et-pair* (i.e., *et-pair* for which at least one *event-pair* exist in L).

Example: Let us consider a web-logging application that monitors the order of different actions denoted as A, B, and C, for two users. Fig. 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 mentioned above by adding one more to extract business process constraints from the indexed log files.

4 Pattern Extraction on top of SIESTA

In this work, 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 [14,6] . The patterns in Declare fall into three distinct categories: existence and unordered relation, position, and ordered relations [14,6]. Table 1 presents various Declare patterns that can be applied to a sequence $S = \langle ev_1, \ldots, ev_n \rangle$, considering the event types a and b (i.e., $a, b \in A$) [7]. This table employs the definition below.

Definition 5. ***(Event type occurrences)*** *For a trace $\sigma = \langle ev_1, ev_2, \ldots, ev_m \rangle$, event occurrences denoted as $O_a^\sigma = [ev_i | ev_i.type = a]$, where $a \in A$ and $ev_i \in \sigma$.*

We now focus on SIESTA's query processing extension to support the efficient extraction of Declare patterns. In our implementation, we allow the support

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

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)	$\forall ev_i \| ev_i.type = a\ \exists ev_j \| ev_j.type = b, i < j$	if a occurs, then b occurs after a
precedence(a,b)	$\forall ev_j \| ev_j.type = b\ \exists ev_i \| ev_i.type = a,\ i < j$	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)	$\forall ev_i \| ev_i.type = a\ \exists ev_j \| ev_j.type = b, i < j$	a can never occur before b
alternate response(a,b)	$\forall ev_i \| ev_i.type = a\ \exists ev_j \| ev_j.type = b \wedge$ $\exists ev_k \| ev_k.type \neq a,\ i < k < j$	Each time a occurs, then b occurs afterwards, before a recurs
alternate precedence(a,b)	$\forall ev_j \| ev_j.type = b \exists ev_i \| ev_i.type = a \wedge$ $\exists ev_k \| ev_k.type \neq b,\ i < k < j$	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)	$\forall ev_i \| ev_i.type = a,\ ev_{i+1}.type = b$	Each time a occurs, then b occurs immediately afterwards
chain precedence(a,b)	$\forall ev_i \| ev_i.type = b,\ ev_{i-1}.type = a$	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 in never immediately followed by b

Table 1: Declare Patterns

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.

Algorithm 1 Extract ordered relation patterns

Input *pattern-type, support*
Output *Ordered Relation Patterns*
1: **for every** (a,b) **in** IndexTable **do**
2: R[a,b] ← IndexTable[a,b]
3: **for every** (a,b) **in** R **do**
4: R[a,b] ← R[a,b] ⋈ SingleTable[a] ⋈ SingleTable[b]
5: C ← **count_occurrences**(R, *pattern-type*)
6: X ← **extract_total_occurrences**(SingleTable)
7: **return filter**(C, X, *support*, *pattern-type*)

4.1 Ordered Relation Patterns

As discussed in Sec. 3, 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.

For an *et-pair* (a, b), the IndexTable[a,b] contains only traces that include this pair, i.e. where an event of type b occurs after an event of type a (line 1-2). However, there might be multiple instances of both event types a and b in a trace. For example, consider a trace $\langle a_1, a_2, b_1, b_2\rangle$. Following the event pair extraction presented in Sec. 3, only the pair $\langle a_1, b_1\rangle$ will be 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 the instances, where event b appears after event a, we require all occurrences of both a and b in the trace to be present. Specifically, for the alternate patterns, we require information about any events of type a or b that occur between a possible occurrence of (a,b). Therefore, we join the IndexTable[a, b] with the records from SingleTable[a] and SingleTable[b] (line 3-4). Alternatively, IndexTable[a, a (resp. b, b)] could have been employed. Consider the trace $\langle a_1, a_2, b_1, a_3\rangle$. After implementing the modification in the previous section, IndexTable [a,a] will include the event-pairs (a_1,a_2) and (a_2,a_3) (a_2 is part of both *event-pairs*). Consequently, when we retrieve IndexTable [a,a], all occurrences of event type a will be available. Since utilizing either IndexTable[a, a] or SingleTable[a] will yield the same results, we opt for the one with the smaller overall size, which depends on the case. After joining these three lists, we will have all occurrences of both event types in the traces where at least one occurrence of the pair (a, b) exists. For a pair (a, b), the resulting structure, denoted as R[a,b], will have the following format: $[(trace_{id}, O_a^\sigma, O_b^\sigma)]$, where O_x^σ is a list containing all the occurrences of event type x in the trace with the id equal to $trace_{id}$ (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 5). The function operates as follows: for a pair (a, b), we iterate through O_a^σ and count how many of these events have an event in O_b^σ that occurred after them, where $O_a^\sigma, O_b^\sigma \in R[a, b]$. For the precedence($a$, b) pattern, we do the opposite: we iterate through O_b^σ and count how many of these events have an event in O_a^σ 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. 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 the 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 6). Finally, the function *filter* (line 7) 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 records from the IndexTable that correspond to the pairs (A,B), (A,C), and (B,C). Next, in lines 3-4, we combine the previous records with the corresponding records from the SingleTable, resulting in the following output: R[A,B]: [(1,[1,2],[3]), (2,[1],[2])], R[A,C]: [(2,[1],[3])], and R[B,C]: [(2,[2],[3])]. In line 5, 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 6 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 7).

4.2 Existence and Unordered Relation Patterns

All existence patterns can be extracted using three structures:

S: For each event type x, this structure contains a list of tuples ($trace_{id}$, O_x^σ), 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**.

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.

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.

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:

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

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 the $|OA|$ we simply need to subtract from the number of unique traces where a appears the number of unique traces where both a and b appear together ($|OA| = U[a] - |I[a,b] \cup I[b,a]|$). 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., $occurrences = |N| + |I[a,b]UI[b,a]| = total\_traces - U[a] - U[b] + 2 \times |I[a,b]UI[b,a]|$.

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

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

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

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.

4.3 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).

Datasets	Events	Traces	Event Types	Length per trace		
				Mean	Min	Max
BPI_2017	1,202,267	31,509	26	38.1	10	180
BPI_2018	2,514,266	43,809	41	57.3	24	2973
BPI_2019	1,595,923	251,734	42	6.3	1	990

Table 2: Dataset characteristics

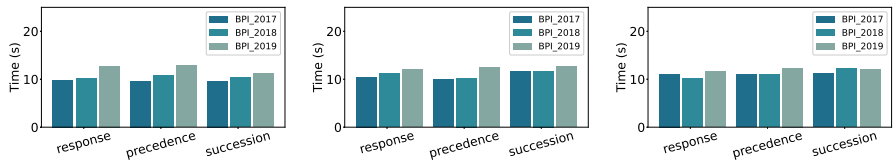
5 Evaluation

Datasets. Among the event logs made available by the Business Process Intelligence (BPI) Challenges, we selected the ones of 2017, 2018, and 2019, because of their significant number of events and average trace length compared to others.[5] Table 2 presents the key characteristics of these datasets, including the number of traces and total events in the log, as well as the minimum, maximum, and mean number of events per trace. However, for our last scalability experiment, we consider the BPI_2011 dataset, which originally contains fewer events (150K), but 624 distinct event types, and we cloned its traces to obtain datasets with increasing number of events (up to 10M).

Competitors. We evaluate the proposed solution against 5 other approaches in the literature. The first one, is the Declare Miner implemented in RuM[6] (version 0.7.2). The second competitor is MINERful; instead of also using RuM, we experimented with a more efficient implementation.[7] The third competitor is the Python library Declare4Py. [8] The remaining two approaches leverage the MATCH_RECOGNIZE (MR) SQL operator and a graph-based data representation. Specifically, for the MR approach, we employed the Trino implementation (version 429) with a PostgreSQL database. In the graph-based approach, we encoded the event log as a graph, which is stored in a Neo4j database (version 5.12). Subsequently, we executed queries using the Cypher graph query language to extract the Declare patterns.

Experiments. We employed a single machine running Ubuntu 20.4, equipped with 32GB of RAM and 8 cores (16 threads) at 3.8 GHz. In the case of SIESTA, 7GB of RAM were allocated to the database, while the remaining resources were dedicated to the preprocessing component and the query processor. The other approaches were given unrestricted access to the available resources. Additionally, for all queries across all approaches, we set the support threshold to 90% (the default value for RuM). Note that, to allow for a fair comparison, we do not test running the techniques over a cluster of machines, because only SIESTA can benefit from massive parallelism. Our competitors can exploit the available cores and resources of the single machine we have used, e.g., RuM spawns multiple

[5] `https://data.4tu.nl/`
[6] `https://rulemining.org/`
[7] `https://github.com/cdc08x/MINERful`
[8] `https://github.com/ivanDonadello/Declare4Py`



(a) Simple (b) Alternate (c) Chain

Fig. 2: Response times for the extraction of the various ordered relation patterns.

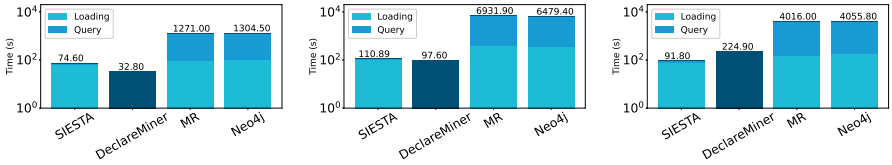
threads. All our experiments, along with detailed instructions for deploying and executing our approach, are publicly available[9].

Initially, we assess the response time of our solution for ordered relation patterns. The indexing times were 59.3, 94, and 64.2 seconds for BPI_2017, BPI_2018, and BPI_2019, respectively. The response times for various ordered relation patterns are illustrated in Fig. 2. Notably, there is no significant difference in response times between the simple, alternate, and chain patterns. This similarity arises because the most time-consuming operation, i.e., fetching and joining records from IndexTable and SingleTable, is common to all three. Furthermore, response and precedence patterns exhibit identical response times across all three settings. Intriguingly, the succession pattern, requiring the calculation of both response and precedence, takes less time than the combined response times of individual patterns. This efficiency is due to fact that the most time-consuming operations occur only once. Remarkably, it takes less than 12 seconds to extract response or precedence patterns from all 3 datasets.

Next, we compare the response times of SIESTA in extracting a single ordered relation pattern, specifically the precedence relation, against the other techniques. The results, encompassing both loading and pattern extraction times, are illustrated in Figure 3. For SIESTA, the loading time is equivalent to the index building (or preprocessing) time. Note that RuM only provides a consolidated time for both data import and extraction; hence, this combined time is presented. It is also important to mention that Declare4Py is not included in this comparison since it does not discern specific pattern types but rather extracts the entire set of Declare patterns simultaneously. The same applies to MINERful. A comparison involving the extraction of the complete pattern set is presented later in this section.

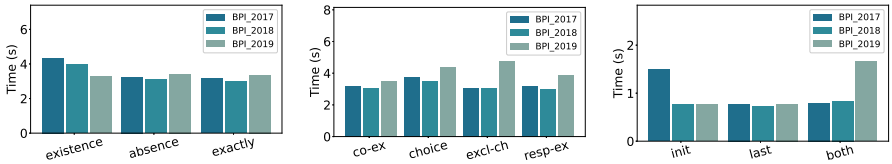
The results show that both SIESTA and RuM's Declare Miner have superior response times compared to MR and Neo4j. This is because the latter approaches lack a mechanism to prune the search space, requiring a complete dataset scan to validate each pattern. Due to the substantial difference in response times, approaching nearly two orders of magnitude for the largest dataset (BPI_2018), we will exclude them from consideration in the upcoming experiments. Declare Miner and SIESTA demonstrate comparable performance, with Declare Miner

[9] `https://anonymous.4open.science/r/SIESTA-BPM-experiments-F1C7`



(a) BPI_2017 (b) BPI_2018 (c) BPI_2019

Fig. 3: Response times for the extraction of only the precedence patterns from the real-world datasets.



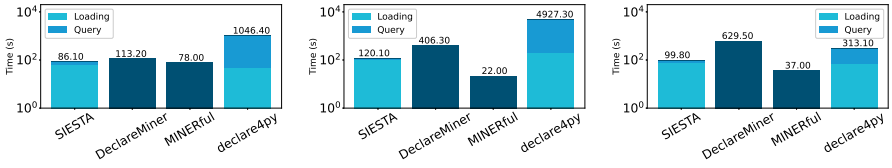
(a) Existence (b) Unordered Relation (c) Position

Fig. 4: SIESTA's Response times for the existence, unordered relation, and position patterns from the real-world datasets.

achieving better response times in the BPI_2017 and BPI_2018 datasets, while SIESTA exhibits faster response times in BPI_2019. According to Table 2, the latter dataset has the highest number of traces, likely contributing to Declare Miner's longer response times. Moreover, as evident from Figure 3, over 95% of SIESTA's time is attributed to building the indices. The benefits of these indices cannot be fully assessed from the extraction of a single pattern. Therefore, we will evaluate later the performance of extracting the complete set of patterns multiple times, for different support thresholds.

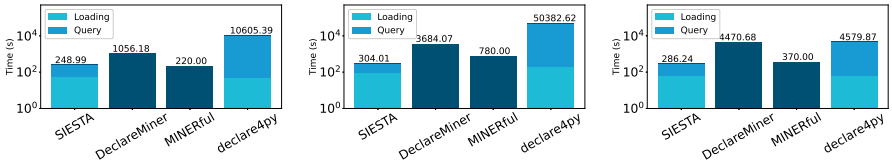
Figure 4 presents the response times for the existence, unordered relation, and position patterns. Despite the size of the dataset, all patterns are extracted in less than 6 seconds, highlighting the effectiveness of utilizing SIESTA's indices for extracting Declare patterns.

Next, we compare the extraction time for the complete set of Declare patterns across various approaches. Two distinct scenarios were evaluated: first, data were imported into each system, and a single extraction of the complete set of rules was performed with the support threshold set to 0.9. In the second scenario, the data were imported again, and multiple pattern extraction processes were executed for different support values. Specifically, the pattern extraction was executed 10 times with the support threshold ranging from 0.8 to 0.89. The results for the two experiments are depicted in Fig. 5 and Fig. 6, respectively. Fig. 6 shows that Declare Miner takes up to 15.6 times longer in the repeating pattern extraction tasks, while MINERFul is overall 1.63 times slower.



(a) BPI_2017 (b) BPI_2018 (c) BPI_2019

Fig. 5: Response times for the extraction of the complete set of patterns from the real-world datasets (support=0.9).



(a) BPI_2017 (b) BPI_2018 (c) BPI_2019

Fig. 6: Response times for the extraction of the complete set of patterns from the real-world datasets for 10 support thresholds (10 queries).

#Events	SIESTA-Cass	SIESTA-S3	MINERful	Declare Miner
1K	8.7	26.2	3.1	**1.4**
10K	11.8	44.1	13.1	**7.7**
100K	**42.1**	108.4	74.7	108.9
1M	**167.6**	314.5	373.7	631.0
10M	-	**2695.1**	3222.0	-

Table 3: Response time (secs) for various instances of the BPI_2011 dataset

In summary, our approach exhibits superior performance across all three datasets, providing responses that reach up to 28 times faster than Declare4Py. RuM's Declare Miner ranks third in response times and confirms our earlier assumption that its performance depends on the number of traces in the log file rather than the number of events. Even in a less favorable scenario for SIESTA, where only one query is performed, and thus the indices are not fully utilized, SIESTA outperforms these competitors. With all three datasets considered, the process of extracting the Declare patterns was executed in under 3 minutes. However, for isolated executions, MINERful seems to have higher performance (see Figure 5), but the overheads of our solutions are quickly outweighed by its performance gain even after only 10 repeated queries, as shown in Figure 6.

Finally, to further assess the scalability of our proposed approach, Table 3 displays the results obtained with datasets artificially generated from the BPI_2011

event log, utilizing both Cassandra and S3 databases.[10] Our solution, especially when implemented with Cassandra, consistently outperforms competitors for medium to large datasets (100K-1M events). However, in a larger scenario of logs containing 10M events, while the S3-based solution still surpasses MINERful, Cassandra fails to complete due to its substantial resource requirements for processing a large number of read and write requests, leading to insufficient memory on the commodity machine; normally, in such cases, Cassandra is deployed on a dedicated server, but we did not want to change the computational infrastructure to keep the experiment fair. On the other hand, S3 demands minimal resources but introduces higher overhead when writing SIESTA's indices into immutable distributed files (in parquet format) and lacks advanced built-in capabilities, such as caching. Nonetheless, this overhead is offset by the high compression rate for large datasets.

6 Conclusion and Future Work

This work has shown how existing general-purpose big data analytics frameworks can be exploited effectively in process mining, 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. The proposed solution successfully addressed the challenge of handling large event logs and extracting business process Declare constraints. The evaluation results show that our solution outperforms other state-of-the-art methods, particularly when the constraint extraction process must be repeated, e.g., for different support thresholds.

Besides providing a best-in-class declarative process constraint extraction framework, this work has illustrated the seamless extensibility of SIESTA, alleviating the need for ad-hoc pattern extraction tools specifically for BPM purposes. In the future, we aim to extend our implementation by introducing a component for conformance checking and temporal compliance [3] of the indexed traces based on a list of rules. This will enable the identification of outlying traces and changes in process execution. Furthermore, as outlined by [11], 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.

References

1. van der Aalst, W.M., Carmona, J.: Process Mining Handbook. Springer (2022)
2. Alman, A., Ciccio, C.D., Haas, D., Maggi, F.M., Nolte, A.: Rule mining with rum. In: Proc. ICPM. pp. 121–128. IEEE (2020)
3. Augusto, A., Awad, A., Dumas, M.: Efficient checking of temporal compliance rules over business process event logs. CoRR **abs/2112.04623** (2021)

[10] Declare4py is omitted because it cannot complete the process if thresholds are reduced to avoid falsely pruning events.

4. Back, C.O., Slaats, T., Hildebrandt, T.T., Marquard, M.: Discover: accurate and efficient discovery of declarative process models. International Journal on Software Tools for Technology Transfer **24**(4), 563–587 (2022)
5. van Beest, N., Groefsema, H., García-Bañuelos, L., Aiello, M.: Variability in business processes: Automatically obtaining a generic specification. Information Systems **80**, 36–55 (2019)
6. De Smedt, J., Deeva, G., De Weerdt, J.: Mining behavioral sequence constraints for classification. IEEE TKDE **32**(6), 1130–1142 (2019)
7. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Trans. Manag. Inf. Syst. **5**(4), 24:1–24:37 (2015)
8. Donadello, I., et al.: Declare4Py: A python library for declarative process mining. In: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2022. vol. 3216, pp. 117–121 (2022)
9. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The prom framework: A new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) Applications and Theory of Petri Nets 2005. pp. 444–454 (2005)
10. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018)
11. Elgammal, A., Turetken, O., van den Heuvel, W.J., Papazoglou, M.: Formalizing and appling compliance patterns for business process compliance. Software & Systems Modeling **15**, 119–146 (2016)
12. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. J. Data Semant. **10**(1-2), 109–141 (2021)
13. Maggi, F.M., Bose, R.J.C., van der Aalst, W.M.: Efficient discovery of understandable declarative process models from event logs. In: CAiSE. pp. 270–285 (2012)
14. Maggi, F.M., Mooij, A.J., Van der Aalst, W.M.: User-guided discovery of declarative process models. In: IEEE CIDM. pp. 192–199. IEEE (2011)
15. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Proc. CAiSE. pp. 270–285 (2012)
16. Maggi, F.M., Di Ciccio, C., Di Francescomarino, C., Kala, T.: Parallel algorithms for the automated discovery of declarative process models. Inf. Syst. **74**(P2), 136–152 (2018)
17. Mavroudopoulos, I., Gounaris, A.: SIESTA: A Scalable InfrastructurE of Sequential paTtern Analysis. IEEE Transactions on Big Data pp. 1–16 (2022)
18. Mavroudopoulos, I., Gounaris, A.: A comprehensive scalable framework for cloud-native pattern detection with enhanced expressiveness. CoRR **abs/2401.09960** (2024). `https://doi.org/10.48550/ARXIV.2401.09960`, `https://doi.org/10.48550/arXiv.2401.09960`
19. Pesic, M., Schonenberg, H., Van der Aalst, W.M.: Declare: Full support for loosely-structured processes. In: Proc. EDOC. pp. 287–287 (2007)
20. Schönig, S., Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: Proc. CAiSE. pp. 290–305 (2016)
21. Westergaard, M., Stahl, C.: Leveraging super-scalarity and parallelism to provide fast declare mining without restrictions. In: BPM Demos. pp. 31–35 (2013)
22. Zaki, N.M., Helal, I.M., Hassanein, E.E., Awad, A.: Efficient checking of timed ordered anti-patterns over graph-encoded event logs. In: International Conference on Model and Data Engineering. pp. 147–161. Springer (2022)