Efficient and Customisable Declarative Process Mining with SQL

Stefan Schönig[1], Andreas Rogge-Solti[1], Cristina Cabanillas[1], Stefan Jablonski[2], and Jan Mendling[1]

[1]Vienna University of Economics and Business, Austria
`{firstname.surname}@wu.ac.at`
[2]University of Bayreuth, Germany
`{stefan.jablonski}@uni-bayreuth.de`

Abstract. Flexible business processes can often be modelled more easily using a declarative rather than a procedural modelling approach. Process mining aims at automating the discovery of business process models. Existing declarative process mining approaches either suffer from performance issues with real-life event logs or limit their expressiveness to a specific set of constaint types. Lately, RelationalXES, a relational database architecture for storing event log data, has been introduced. In this paper, we introduce a mining approach that directly works on relational event data by querying the log with conventional SQL. By leveraging database performance technology, the mining procedure is fast without limiting itself to detecting certain control-flow constraints. Queries can be customised and cover process perspectives beyond control flow, e.g., organisational aspects. We evaluated the performance and the capabilities of our approach with regard to several real-life event logs.

Keywords: Declarative Process Mining, Relational Databases, SQL

1 Introduction

Process mining is the area of research that embraces the automated discovery, conformance checking and enhancement of process models. All involved techniques are evidence-based, as the input is event logs that comprise a collection of computer recorded information that track the executions of process instances.

Two different types of processes can be distinguished [1]: well-structured routine processes with exactly predescribed control flow and flexible processes whose control flow evolves at run time without being fully predefined a priori. Likewise, two different representational paradigms can be distinguished: procedural models describe which activities can be executed next and declarative models define execution constraints that the process has to satisfy. The more constraints we add to the model, the less possible execution alternatives remain. As flexible

This work was funded by the European Unions Seventh Framework Programme (FP7/2007-2013) grant 612052 (SERAMIS) and the Austrian Research Promotion Agency (FFG) grant 845638 (SHAPE).

processes may not be completely known a priori, they can often be captured more easily using a declarative rather than a procedural modelling approach [2–4]. Declarative languages like *Declare* [5], *Dynamic Condition Response* (DCR) graphs [6] or *Declarative Process Intermediate Language* (DPIL) [7] can be used to represent these models, and tools like DeclareMiner [8], MINERful [9] or DPILMiner [10] offer capabilities to automatically discover such models from event logs. Existing declarative process mining approaches either suffer from performance issues with real-life event logs or limit their search space to a specific and fixed set of constaints to be able to cope with the size of real-life event logs. Both issues are highlighted in current literature, e.g., [9–12]. To the best of our knowledge an approach that is fast and customisable does not exist.

We fill this research gap by introducing a declarative mining approach that works on event data that is stored in relational databases by querying the log with conventional SQL. An overview of the approach is given in Fig 1. Process mining by means of SQL queries turns out to be an integrated and language overspanning solution to process discovery. By leveraging relational database performance technology, e.g., indexes on data columns, it is relatively fast without limiting itself to certain predefined constraints. Queries can be tailored to arbitrary aspects of a process, e.g., control flow as well as organisational issues. Furthermore, the results can be transformed to each of the mentioned process modelling languages. We evaluated the performance using several real-life event logs. Moreover, we demonstrate capabilities, expressiveness and additional insights of our approach by providing a list of queries for discovering commonly used process constraints. All queries are published in a technical report [13]. Further material as well as a screencast are accessible online at `http://sqlminer.kppq.de`. Summing up the contributions of this work, we introduce a *(i) customisable*, *(ii) language independent* and *(iii) performant* declarative process mining technique.

The remainder of this paper is structured as follows: Section 2 describes the input data, fundamentals of declarative process mining and related work. In Section 3 we introduce our approach to discover process constraints using SQL queries. The approach is evaluated in Section 4, and Section 5 concludes the paper.

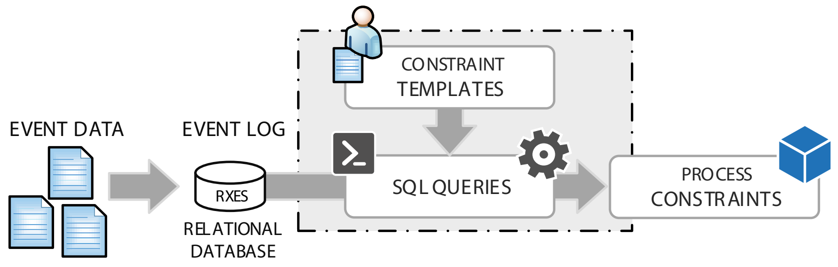

Fig. 1: Overview of the SQL-based process mining approach

2 Background and Related Work

Next, we describe the input data for our approach as well as fundamentals of declarative process modelling and automated discovery of models.

2.1 Storing Event Log Data in Relational Databases

Our process mining approach takes as input *(i) an event log*, i.e., a machine-readable file that reports on the execution of activities during the enactment of the instances of a given process, and optionally *(ii) organisational background knowledge*, i.e., prior knowledge about the roles, capabilities and the assignment of resources to organisational units. In an event log, every process instance corresponds to a sequence (*trace*) of recorded entries, namely, *events*. We require that events contain an explicit reference to the enacted activity. For discovering resource-related aspects we additionally require an explicit reference to the operating resource. Typically, these requirements are met when business processes are enacted by information systems [14].

The standardised, extensible storage format OpenXES was developed for the purpose of storing event data [15]. Lately, with *RelationalXES (RXES)* a relational database architecture for storing event log data has been introduced [16]. The RXES architecture uses a database to store the event log where traces and events are represented by tables with identifiers (IDs). RXES provides a full implementation of all OpenXES interfaces using the database as a backend. The database schema used in RXES allows for a significant reduction of redundancy by storing frequently occurring attributes only once rather than repeating them for every occurrence.

In this paper, we consider an event log to be available according to the RXES architecture and therefore to be stored in a conventional relational database. For readability we use a denormalised event log table like in Table 1, capturing only the attributes *EventID* (unique identifier for each recorded event), *TraceID* (unique identifier for the corresponding trace), *ActivityID* (name of the corresponding activity the event refers to), *Time* (date and time the event has occurred) as well as *Identity* (identifier of the performing resource or person). In the remainder of the paper, we will use the shorthand notation (a,id_x) for indicating an event of activity a that has been executed by an identity id_x. The given events are ordered temporally so that timestamps are not encoded explicitly. The following *example event log* contains four traces comprising events of four different activities and executed by five different resources:

$\langle(a,id_1),(b,id_1),(c,id_2)\rangle$, $\langle(b,id_2),(c,id_2)\rangle$, $\langle(a,id_2),(d,id_4),(c,id_2)\rangle$, $\langle(a,id_5),(a,id_5),(b,id_1),(c,id_3)\rangle$.

In the case that correlations between process execution and resource characteristics should be examined, organisational background information, e.g., in form of an organisational model must also be given in a relational database table. We build upon the generic organisational meta-model defined in [17]. A

Event ID	Trace ID	Activity ID	Time	Identity
1	1	a	2015-11-06 15:31:00	id_1
2	1	b	2015-11-06 15:35:00	id_1
3	1	c	2015-11-06 15:37:00	id_2
4	2	b	2015-11-06 16:22:00	id_2
5	2	c	2015-11-06 16:45:00	id_2
...				

Table 1: Event log excerpt stored in a denormalized relational database table

Identity represents an individual person that can be directly assigned to activities. A Group describes several resources as a whole. A Relation represents the interplay between Identity and Group. For Relations, a RelationType specifies its interpretation. For instance, an organisation may have three Groups (Professor, Student, Admin), five Identities (id_1, id_2, id_3, id_4 and id_5) and the following four Relations of RelationType "role" describing the roles that are assigned to each resource:

(id_1,role,Student), *(id_2,role,Professor)*, *(id_3,role,Professor)*, *(id_4,role,Admin)*, *(id_5,role,Student)*.

This knowledge can help us to identify resource allocation rules as well as control flow rules that only apply to certain groups.

2.2 Fundamentals of Declarative Process Mining

In the following, we describe the basic aspects of declarative process modelling and the principle of automated discovery of such models from event log data.

Declarative Process Modelling Languages During the last decade several declarative process modelling languages have been developed [5–7]. These languages are based on so-called *constraint templates*. A constraint template captures frequently occurring relations and defines a particular type of constraint. Templates have formal semantics specified through logical formulae and are equipped either with user-friendly graphical representations (e.g., in Declare [5]) or macros in textual languages (e.g., in DPIL [7]) that make the model easier to understand. A constraint template consists of placeholders, i.e., typed variables. It is instantiated by providing concrete values for these placeholders.

Concrete constraints are rules constraining, e.g., the execution of activities. For example, *Response(a,b)* of the Declare language is a constraint on the activities a and b, forcing b to be executed eventually if activity a was performed before. The *ChainResponse(a,b)* constraint is stronger and forces activity b to be executed *directly* after activity a was executed. Some languages allow in addition to control-flow aspects the definition of constraints concerning the assignment of certain resources to activites. *RoleBasedAllocation(a,r)* of the DPIL language is a constraint on activity a, demanding a, if executed, to be performed by a person in a specific role r.

As shown, constraint templates can be categorized into different groups depending on the process perspective they concern or their cardinality, i.e., the number of activities affected. For example, instances of the *Response(A,B)* template are pure control flow constraints that describe the temporal relation between two activities. On the other hand, the *RoleBasedAllocation(A,R)* template refers to the resource perspective and constrains the resources that are assigned to one certain activity.

Declarative Process Mining For almost each declarative process modelling language a corresponding mining approach exists. The first approach to extract declarative constraints from event logs introduced by Maggi et al. is the *DeclareMiner* [8]. Here, the user can select from a set of predefined Declare constraint templates the ones to be discovered. The system then generates all possible constraints by instantiating the chosen set of constraint templates with all possible combinations of occurring process elements provided in the event log. For example, the *Response(A,B)* template consists of two placeholders for activities. Assuming that $|A|$ different activities occur in the event log, $|A|^2$ *constraint candidates* are generated. All the resulting candidates are subsequently checked w.r.t. the event log. Additional mining parameters like PoE (Percentage of Events) or PoI (Percentage of Instances) are used to distinguish between valid and non-valid constraint candidates. Maggi et al. propose an evaluation of this algorithm with the adoption of a two-phase approach [11]. During the first phase, frequent sets of correlated activities are identified. The candidate constraints are only generated on the basis of these activities. In the second phase, the candidates are then checked in the same way as in [8]. Additionally, there are post-processing approaches that aim at simplifying the resulting Declare models in terms of, a.o., redundancy elimination [18, 9] and disambiguation [19]. In essence, the focus of these approaches is control flow with extensions to cover data dependencies [20]. The *DPILMiner* [10] proposes a declarative mining approach to incorporate the resource perspective and to mine for a set of predefined resource assignment constraints. All the mentioned approaches suffer from performance issues w.r.t. real-life event logs. Furthermore, the set of constraint templates to be analysed is predefined, i.e., cannot be customised by the user.

Efficient algorithms to discover Declare models are presented in [21, 12]. Westergaard presents the *UnconstrainedMiner* [12], whose outstanding performance is obtained by constraint checking parallelisation and by relying on efficient data structures. The *MINERful* approach [9] that has been implemented [21] in the ProM framework has shown to be the most efficient algorithm to discover control-flow constraints using the Declare language. Both approaches, however, are limited to discover control-flow constraints of the Declare language.

Mining Metrics Checking constraint candidates provides for every constraint candidate the number of satisfactions in the event log. The constraint *Response(a,b)*, e.g., is satisfied three times in the example event log since in three cases of an occurrence of a, b eventually follows in the same trace. Based on the number of satisfactions two metrics *Support* and *Confidence* are calculated that express the probability of a constraint to hold in the process. Constraints are considered valid, if their Support and Confidence measures are above a user defined threshold. In literature, there are two different definitions of support and confidence in the context of process mining. For our approach we adopt the most recent definition by Di Ciccio et al. [9] where the metrics are defined as follows:

- **Support:** It is the number of fulfilments of a constraint divided by the number of occurrences of the condition of a constraint. In the example log, the support of *Response(a,b)* is 0.75, as 3 a's out of 4 fulfil the constraint. In case of constraints that do not depict implications like *Existence* constraints in Declare, the Support is defined as the number of fulfilments divided by the number of traces in the log.
- **Confidence:** It is the product of the support and the fraction of traces in the log where the condition (implications) or the constrained activity (not implications) occurs. The confidence of *Response(a,b)* is $0.75 \cdot 0.75 = 0.5625$, since condition a occurs in 3 traces out of 4.

The definitions above show that for discovering a certain constraint three different constraint specific values need to be extracted from event logs: *(i) the number of occurrences of the condition of the constraint*, *(ii) the number of fulfilments of the constraint*, and *(iii) the fraction of traces in the log where the condition holds.*

3 Declarative Process Mining with SQL

With RXES a standardised architecture for storing event log data in relational databases was introduced. In this section we show that it is possible to extract relevant process knowledge from event logs stored in relational tables by applying conventional database queries without any parsing or data conversion. The Structured Query Language (SQL) is a declarative language designed for managing data held in a relational database. It is based upon relational algebra. For space reasons we cannot explain language details and therefore refer to one of the available SQL handbooks [22].

In the following we introduce a procedure to map constraint templates from various declarative process modelling languages to SQL queries. First, we describe the general query assembly by means of the *Response* constraint template. Subsequently, we describe the mapping of a more complex control flow constraint template. Finally, we show that it is possible to map and discover customised constraints involving different process perspectives apart from control flow.

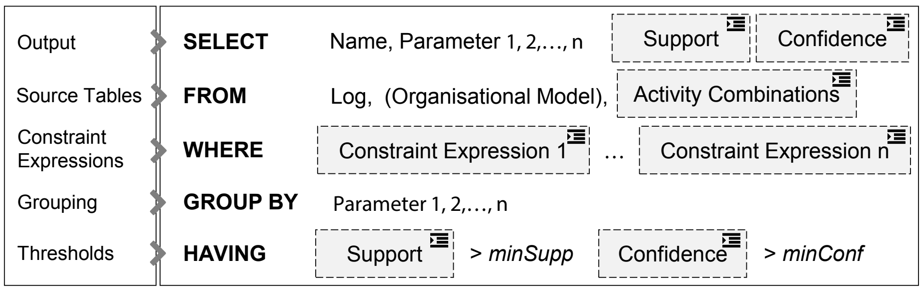

Fig. 2: SQL query assembly for querying declarative constraint templates

3.1 General Query Assembly

Fig. 2 depicts the basic structure of an SQL query that discovers all constraints of a certain template under consideration of two thresholds *minSupp* and *minConf*. Here, subqueries are marked with a symbol on the upper right corner. We describe the different SQL directives in the order they get executed.

We show the approach with an example, namely, the *Response(A,B)* constraint template, which aims at discovering all the activity combinations (a,b) where b is forced to be executed if a was completed at some point before. As depicted in Fig. 2, the query for discovering *Response* constraints is organised as follows:

```
SELECT  'response', A, B, [Support], [Confidence]
FROM Log l1, [ActivityCombinations] c
l1.Activity = c.A
AND
EXISTS(SELECT *
       FROM Log l2
       WHERE l2.Activity = c.B AND l2.Instance = l1.Instance AND
             l2.Time > l1.Time)
GROUP BY c.A, c.B
HAVING [Support] > minSupp AND [Confidence] > minConf
```

FROM clause: Here, the data source tables are joined together, i.e., the table of the analysed event *Log* where every tuple depicts a single event and, if available, the table of the *OrganisationalModel*. Furthermore, the clause contains a subquery *ActivityCombinations* that provides a table with the activity combinations that should be checked. In case of constraints that comprise two activities like *Response*, this subquery looks as follows:

```
SELECT l1.Activity AS A, l2.Activity AS B
FROM Log l1, Log l2 WHERE l1.Activity != l2.Activity
GROUP BY l1.Activity, l2.Activity
```

Every source table gets an abbreviation assigned to be referable in other clauses, e.g., "l1" for the event log table or "c" for the combination table.

WHERE clause: This clause contains the different constraint expressions that have to hold for activities and their events, i.e., the constraint activation condition as well as their fulfilment requirements. If an expression cannot be derived directly, a corresponding subquery is used. For *Response* constraints the condition activation is any occurrence of an event of the currently selected activity a. In order to be fulfilled the constraint expression has to hold: activity b must occur in the same instance and it must occur eventually after a. Therefore, the WHERE clause for querying *Response* constraints contains a subquery:

```
l1.Activity = c.A
AND
EXISTS(SELECT *
       FROM Log l2
       WHERE l2.Activity = c.B AND l2.Instance = l1.Instance AND
             l2.Time > l1.Time)
```

It tests if at least one event of an activity b exists that is executed after the currently observed event of a. In case all logical terms in the clause evaluate to true, the currently observed tuple depicts one fulfilment of the constraint. The resulting set of tuples depicts all fulfilments of the analysed constraint template.

GROUP BY clause: After deriving the fulfilments, the tuples are grouped by the set of parameters of the constraint template. Hence, in the query to discover *Response* constraints, we group w.r.t. the two parameters referring to activities A and B.

HAVING clause: As described in Section 2.2 the number of fulfilments of each constraint needs to be computed in order to calculate the support and confidence values. After grouping, the number of tuples corresponding to a certain parameter combination can be extracted using the SQL aggregate function COUNT(*). In addition, a subquery computes the number of occurrences of the condition of the constraint. This way, the *Support* value of each constraint can be derived. In case of *Response* constraints the condition is depicted by the number of occurrences of the currently selected activity a. The SQL expression for calculating the *Support* of *Response* constraints is given as:

```
COUNT(*)  /  (SELECT COUNT(*) FROM Log WHERE Activity = A)
```

The *Confidence* of each parameter combination can be calculated in a similar way. Recall that *Confidence* is defined as the product of *Support* and the fraction of traces in the log where the condition occurs. Therefore, in case of *Response*, we calculate the fraction of the outcome of two subqueries to *(i)* extract the number of instances where a occurs at least once and *(ii)* to extract the number of traces in the log.

```
Support * ((SELECT COUNT(*) FROM
(SELECT Instance FROM Log WHERE Activity = A GROUP BY Instance) t ) /
(SELECT COUNT(*) FROM (SELECT Instance FROM Log GROUP BY Instance) t2 ))
```

ID	**Name**	**A**	**B**	**Support**	**Confidence**
1	Response	a	b	0.75	0.5625
2	Response	a	c	1	0.75
3	Response	b	c	1	0.75
4	Response	d	c	1	0.25

Table 2: Result table of the *Response* query w.r.t. the example log

The resulting values of both queries can then be filtered by user-defined thresholds.

SELECT clause: In the last step the query output is selected, i.e., the parameter combination and its corresponding *Support* and *Confidence* values. The result set contains tuples for each parameter combination that fulfils the constraint under consideration of the given thresholds. Table 2 results from applying the *Response* query to the example event log of Section 2.2 with the thresholds *minSupp=0.7* and *minConf=0.2*.

3.2 Mining Complex Control-Flow Constraints

More complex control flow constraint templates can be mapped to SQL queries by adding the additional constraint expressions of the constraint template to the WHERE-clause of the query. Consider for example the *ChainResponse(A,B)* constraint template of the Declare language which states that an activity *b* is always executed *directly* after an activity *a* has been completed. In addition to the fulfilment requirements of a *Response* constraint, a *ChainResponse* requires that *b* has to follow *a* without any other activity in-between. This additional requirement is implemented by adding another subquery to the WHERE-clause:

```
[Response Constraint Expressions Subquery]
AND
NOT EXISTS(SELECT * FROM Log l2, Log l3
           WHERE l3.Instance = l1.Instance AND l2.Instance = l1.Instance
           AND l2.Activity = c.B
           AND l3.Time > l1.Time AND l3.Time < l2.Time)
```

The subquery provides all events that temporally took place in-between the currently selected event of an activity *a* and any event of an activity *b* in the same instance. The constraint is only fulfilled for a certain activity combination (*a*,*b*), if there exists no event of any activity between any occurrence of *a* and *b*. Other constraint templates of the Declare language, like *AlternateResponse*, *Precedence* or *NotSuccession* can be mapped in an analogous manner. Subsequently, the result set can be post-processed with existing pruning methods for declarative process mining, like pruning based on constraint hierarchies [23].

3.3 Customised Queries including Additional Perspectives

In this section, we demonstrate how queries can be customised by adding aspects of further process perspectives. First, we focus on the organisational perspective, i.e., we describe how resource assignment contraints can be discovered by means of SQL queries. Second, we show how queries can be customised in order to analyse the interplay of different perspectives, specifically to discover the influence of resources on the control flow of the process [10]. We explain the functionality with example constraints.

Resource Assignment Constraints. Resource assignment in business processes is extensively discussed by the workflow resource patterns. These patterns capture the various ways in which resources are represented and utilised in processes [24]. As an example, we explain the SQL query to extract role-based resource assignment constraints, i.e., that a certain activity can only be executed by resources assigned to a certain role. This constraint type is captured by the *RoleBasedAllocation(A,R)* template in the DPIL language and consists of parameters for activities *A* and roles *R*. Here, we assume organisational information to be available in a relational table *Relations*. Without loss of generality *Relations* has the attributes *Resource*, *RelationType* and *Group*. The FROM, WHERE and GROUP BY clauses of the query are then given as follows:

```
FROM Log l, Relations r1, [ActivityCombinations] c
WHERE l.Activity = c.A AND r1.RelationType = 'role'
AND l.Resource = r1.Resource AND
NOT EXISTS(SELECT *  FROM Log l2, Relation r2
           WHERE l2.Resource = r2.Resource AND r2.RelationType = 'role'
           AND l2.Activity = l.Activity AND NOT r2.Group = r1.Group)
GROUP BY c.A, r1.Group
```

In addition to the event log and the activity combinations we also join the table with all relations according to the organisational model in the FROM clause. The constraint is only fulfilled for a certain event of an activity a if there exists no event of a that has been performed by a resource with a different role. The fulfilments are then grouped by every occurring activity and role combination. Since the condition of the constraint is like in case of the *Response* constraints the occurrence of activity a, the SELECT and HAVING clauses are identical. Applying the query with the thresholds *minSupp=0.7* and *minConf=0.2* to the example event log and organisational model described in Section 2.1 results in

ID	Name	A	Role	Support	Confidence
1	RoleBasedAllocation	d	Admin	1	0.25
2	RoleBasedAllocation	c	Professor	1	1

Table 3: Result table of the *RoleBasedAllocation* query w.r.t. the example log

a set of two tuples as shown in Table 3. The constraints express that activity *d* has always been performed by a resource with role *Admin* and activity *c* by a resource with role *Professor*, respectively.

Cross-Perspective Constraints. Resource allocation can also have effects on the execution order of activities. These rules are called "cross-perspective" [10]. They can be found in different application areas, e.g., in cases where the execution order of certain process steps is bound to conditions that hold only for certain resources. A *RoleBasedResponse(A,B,R)* template, e.g., represents situations in which an activity must be executed after another one for specific organisational roles but not for others. These types of contraints cannot be discovered with pure control-flow mining approaches but require a customisation to combine the control flow and organisational perspectives. In order to extract control flow dependencies between activities that only hold for resources with a certain role, the subquery for calculating the *Support* value of *RoleBasedResponse* query needs to be given as follows:

```
COUNT(*)  /  (SELECT COUNT(*) FROM Log l2, Relation r2
WHERE r2.RelationType = 'role' AND l2.Resource = r2.Resource
AND Activity = c.A AND r2.Group = r1.Group)
```

Here, the number of fulfilments of the constraint is divided by the number of occurrences of events of activity *a* that have been performed by resources with a certain role. Applying the customised query with the thresholds *minSupp=0.7* and *minConf=0.2* to the example event log and the organisational model of Section 2.1 the resulting constraints are given in Table 4. When comparing the discovered constraints with the *Response* constraints of Table 2, we can see that in general activity *b* does not have to be executed after *a* in every case (*Support=0.75*). However, when activity *a* has been performed by a Student, *b* followed eventually in every case (*Support=1*). The results of the customised query give a more fine-grained resolution of constraint satisfaction w.r.t. the roles of performing resources.

Note that the higher flexibility in the constraints (e.g., ternary versus binary constraints) increases the number of possible rule candidates exponentially in

ID	Name	A	B	Role	Support	Confidence
1	RoleBasedResponse	a	b	Student	1	0.25
2	RoleBasedResponse	a	c	Professor	1	0.25
3	RoleBasedResponse	a	c	Student	1	0.25
4	RoleBasedResponse	a	d	Professor	1	0.25
5	RoleBasedResponse	b	c	Professor	1	0.25
6	RoleBasedResponse	b	c	Student	1	0.5
7	RoleBasedResponse	d	c	Admin	1	0.25

Table 4: Result table of the *RoleBasedResponse* query w.r.t. the example log

the number of parameters. This curse of dimensionality can lead to overfitting issues, i.e., the constraint model allows us to capture rules that reflect noise rather than real patterns. Nevertheless, this problem is mitigated by the use of a higher confidence threshold for the constraints. That way, we make sure that we do not incorporate rules that are only triggered a few times. Given enough data samples, however, these more precise constraints can also reach higher confidence levels and in the end yield a more accurate model.

4 Evaluation

The approach has been implemented and tested by means of conventional SQL and a relational database. Furthermore, we implemented a tool that imports event logs given in the XML-based XES format to a relational database table. We evaluate our approach in two phases: first, we measure the performance with respect to publicly available real-life event logs. Second, we assess capabilities and expressiveness of SQL-based process mining (*SQLMiner*).

4.1 Performance Analysis

We measure the performance of the SQLMiner w.r.t. two real-life event logs from the financial[1] and healthcare[2] domains by means of control-flow contraints. Their characteristics are summarised in Table 5. The event logs have been imported into a Microsoft SQL Server 2008 R2 database. We compare the performance to the ProM implementations (Version 6.5.1) of state-of-the-art declarative process mining tools, namely, *DeclareMiner* [11], *MINERful* [21] and the *Unconstrained-Miner* [12]. For all tests we use the default settings of each tool.

We execute the queries of eight commonly known Declare constraint templates: *AlternateReponse*, *Response*, *ChainResponse*, *AlternatePrecedence*, *Precedence*, *ChainPrecedence*, *RespondedExistence* and *NotSuccession*. Note that all

[1] doi: `10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`
[2] doi: `10.4121/d9769f3d-0ab0-4fb8-803b-0d1120ffcf54`

Source	**Activities**	**Traces**	**Events**	**Total Time**	**Miner**
Financial Log	24	13 087	262 200	01:03:47	Dec. Miner
				00:00:17	MINERful
				00:00:14	Unc. Miner
				00:01:08	SQLMiner
Hospital Log	624	1 143	150 291	03:11:12	Dec. Miner
				00:12:28	MINERful
				00:16:45	Unc. Miner
				00:19:30	SQLMiner

Table 5: Performance of SQL-based mining over real-life event logs

these templates have as parameters two activities. Hence, the result tables have the same number and type of columns. Consequently, all the queries can be executed together by connecting them with the SQL UNION operator. All the computation times reported in this section are measured on a Core i7 CPU @2.80 GHz with 8 GB RAM. Table 5 shows the results of our performance tests. We can see that for both logs the *MINERful*[3] and *UnconstrainedMiner* tools reach an outstanding performance due to their efficient and specialised data structures. Nonetheless, the *SQLMiner* performs slower than these tools only within the range of seconds or a few minutes, respectively. The *DeclareMiner* takes for the analysis in both cases more than one, respectively three, hours. Recall that in contrast to the specialised Declare mining tools, SQL queries can be adapted to the analysts needs and refer to all the stored process information of different perspectives. Therefore, our approach depicts a trade-off between customisation and performance.

4.2 Capabilities and Expressiveness

For showing the capabilities of SQL-based process mining, we express commonly known process constraints of different perspectives to SQL queries and test them on real-life event logs. All queries are published in a technical report [13] and are additionally accessible online at `http://sqlminer.kppq.de`. For control flow related constraints, we map and test a list of fourteen frequently used Declare constraints, e.g., *Response* and *ChainResponse* among others defined in [9]. For resource-related aspects (i.e., the organisational perspective) we use the group of so-called *creation patterns* of the workflow resource patterns [24] including, a.o., role-based allocation and separation of duties. The SQLMiner is able to discover six creation patterns, similarly to the DPILMiner [10].

Furthermore, we show the additional insights in process data by applying *(i)* the traditional *Response* query and *(ii)* the cross-perspective *RoleBasedResponse* query to a real-life university business trip management event log[4]. The log contains 2104 events of 10 different activities related to the application and the approval of university business trips as well as the management of accommodations and transfers, i.e., booking hotels and buying transport tickets. The system has been used for 6 months by 10 university employees. In total, there are 128 business trip cases recorded in the log. We configured the queries with the thresholds $minSupp = 0.7$ and $minConf = 0.5$, respectively. The result set was composed of 29 *Response* and also 29 *RoleBasedResponse* constraints. The additional insights are explained by means of the following two extracted constraints that describe the temporal dependency between the activities *Apply for Trip* and *Book flight*:

– *Response*, *Apply for trip*, *Book flight*, *Supp=0.75*, *Conf=0.75*
– *RoleBasedResponse*, *Apply for trip*, *Book flight*, *Student*, *Supp=1*, *Conf=0.75*

[3] The MINERful plugin does not allow to select certain templates to be analysed.
[4] The event log is available for download at *sqlminer.kppq.de*.

We discovered the influence of resources on the execution order of the activities. Although employees usually applied for the trip before they booked the corresponding flight, it was apparently not mandatory. Nonetheless, there are cases in which certain employees already booked the flight without applying for the trip (*Support=0.75*). However, when analysing the ordering of tasks under consideration of performing resources, we extracted that resources with role *Student* always applied for the trip before they booked a flight (*Support=1*). While professors are free to book a flight without an approved application, students mandatorily have to stick to a certain order of tasks.

5 Conclusions and Future Work

In this paper, we introduced an SQL-based declarative process mining approach that analyses event log data stored in relational databases. While existing declarative process mining approaches either suffer from performance issues with real-life event logs or limit their search space to a specific and fixed set of constraints, the SQLMiner approach constitutes a trade-off between performance and customisation capabilities. We showed that it is possible to discover commonly used process constraints by means of conventional SQL queries. Leveraging relational database performance technology the approach is fast without limiting itself to certain predefined constraints. Queries can be customised and comprise process perspectives apart from control flow, such as organisational aspects. We evaluated the performance w.r.t. several real-life event logs. Moreover, we demonstrated capabilities and expressiveness by providing a list of queries for discovering commonly used process constraints.

The approach still has missing aspects, e.g., data-related constraints have not been integrated yet. As future work we plan to extend the SQLMiner to cover further process perspectives as well as to develop a publicly available query tool for declarative process mining on relational databases. Furthermore, understandability of queries will be enhanced by using SQL functions to represent common parts of different queries.

References

1. S. Jablonski, "MOBILE: A modular workflow model and architecture," in *Working Conference on Dynamic Modelling and Information Systems*, 1994.
2. W. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - Research and Development*, vol. 23, no. 2, pp. 99–113, 2009.
3. P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," *Business Process Management Workshops*, pp. 383–394, 2012.
4. R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya, "Declarative business artifact centric modeling of decision and knowledge intensive business processes," in *EDOC*, pp. 151–160, 2011.

5. M. Pesic and W. M. Van der Aalst, "A declarative approach for flexible business processes management," in *Business Process Management Workshops*, pp. 169–180, Springer, 2006.
6. T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti, "Contracts for cross-organizational workflows as timed dynamic condition response graphs," *The Journal of Logic and Algebraic Programming*, vol. 82, no. 5, pp. 164–185, 2013.
7. M. Zeising, S. Schönig, and S. Jablonski, "Towards a Common Platform for the Support of Routine and Agile Business Processes," in *Collaborative Computing: Networking, Applications and Worksharing*, 2014.
8. F. M. Maggi, A. Mooij, and W. van der Aalst, "User-Guided Discovery of Declarative Process Models," in *CIDM*, pp. 192–199, 2011.
9. C. Di Ciccio and M. Mecella, "On the discovery of declarative control flows for artful processes," *ACM Trans. Management Inf. Syst.*, vol. 5, no. 4, pp. 24:1–24:37, 2015.
10. S. Schönig, C. Cabanillas, S. Jablonski, and J. Mendling, "Mining the Organisational Perspective in Agile Business Processes," in *BPMDS*, vol. 214 of *LNBIP*, pp. 37–52, Springer, 2015.
11. F. M. Maggi, J. C. Bose, and W. van der Aalst, "Efficient Discovery of Understandable Declarative Process Models from Event Logs," in *CAiSE*, pp. 270–285, 2012.
12. M. Westergaard, C. Stahl, and H. Reijers, "UnconstrainedMiner: Efficient Discovery of Generalized Declarative Process Models," *BPM Center Report, No. BPM-13-28*, 2013.
13. S. Schönig, "SQL Queries for Declarative Process Mining on Event Logs of Relational Databases," *arXiv preprint arXiv:1512.00196*, 2015.
14. W. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes.* 2011.
15. E. Verbeek, J. Buijs, B. van Dongen, and W. van der Aalst, "XES, xESame, and ProM 6," in *Information Systems Evolution*, pp. 60–75, 2011.
16. B. F. van Dongen and S. Shabani, "Relational XES: data management for process mining," in *CAiSE Forum 2015*, pp. 169–176, 2015.
17. C. Bussler, *Organisationsverwaltung in Workflow-Management-Systemen.* Dt. Univ.-Verlag, 1998.
18. F. M. Maggi, J. C. Bose, and W. van der Aalst, "A Knowledge-Based Integrated Approach for Discovering and Repairing Declare Maps," in *CAiSE*, pp. 433–448, 2013.
19. J. C. Bose, F. M. Maggi, and W. van der Aalst, "Enhancing Declare Maps Based on Event Correlations," in *Business Process Management*, pp. 97–112, 2013.
20. F. M. Maggi and M. Dumas, "Discovering Data-Aware Declarative Process Models from Event Logs," in *Business Process Management*, pp. 1–16, 2013.
21. C. Di Ciccio, M. H. M. Schouten, M. de Leoni, and J. Mendling, "Declarative process discovery with minerful in prom," in *BPM Demos*, pp. 60–64, 2015.
22. J. S. Bowman, S. L. Emerson, and M. Darnovsky, *The Practical SQL Handbook: Using Structured Query Language.* Addison-Wesley Longman Publishing, 1996.
23. C. Di Ciccio, F. M. Maggi, M. Montali, and J. Mendling, "Ensuring model consistency in declarative process discovery," in *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, pp. 144–159, 2015.
24. N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond, "Workflow Resource Patterns: Identification, Representation and Tool Support," in *CAiSE*, pp. 216–232, 2005.