An SQL-Based Declarative Process Mining Framework for Analyzing Process Data Stored in Relational Databases

Francesco Riva[1,2,3(✉)], Dario Benvenuti[4], Fabrizio Maria Maggi[1], Andrea Marrella[4], and Marco Montali[1]

[1] Free University of Bozen-Bolzano, Bolzano, Italy
francesco.riva@unibz.it, {maggi,montali}@inf.unibz.it,
[2] University of Tartu, Tartu, Estonia
[3] Datalane SRL, Verona, Italy
[4] Sapienza University of Rome, Rome, Italy
{d.benvenuti,marrella}@diag.uniroma1.it

Abstract. Recently, the idea of applying process data analysis over relational databases (DBs) has been investigated in the process mining field resulting into different DB schemas that can be used to effectively store process data coming from Process-Aware Information Systems (PAISs). However, although SQL queries are particularly suitable to check declarative rules over traces stored in a DB, a deep analysis of how the existing instruments for SQL-based process mining can be effectively used for process analysis tasks based on declarative process modeling languages is still missing. In this paper, we present a full-fledged framework based on SQL queries over relational DBs for different declarative process mining use cases, i.e., process discovery, conformance checking, and query checking. The framework is used to benchmark different SQL-based solutions for declarative process mining, using synthetic and real-life event logs, with the aim of exploring their strengths and weaknesses.

Keywords: Process Discovery · Conformance Checking · Query Checking · Declarative Process Model · SQL · Relational Database

1 Introduction

The process data recorded by Process-Aware Information Systems (PAISs) is usually stored in multiple and often heterogeneous relational databases (DBs). Several efforts have been done, in the past, in order to solve the data integration

The work of F. Riva was funded by the PRISMA project of the Free University of Bozen-Bolzano and by the Estonian Research Council (PRG1226). The work of D. Benvenuti and A. Marrella was supported by the PNRR MUR project PE0000013-FAIR, the H2020 project DataCloud (Grant number 101016835), and the Sapienza project DISPIPE.

C. Di Francescomarino et al. (Eds.): BPM 2023 Forum, LNBIP 490, pp. 214–231, 2023.
https://doi.org/10.1007/978-3-031-41623-1_13

problem [4], but also in order to store and query process data in relational DBs in an effective and efficient manner [13,31].

In recent years, the process mining community has investigated how DB theory methods can be used to carry on process analysis on the process behavior recorded in a relational DB. Different DB schemas have been developed [13,31], which are suitable to effectively store process data into a DB.

SQL queries are particularly suitable for process mining based on declarative languages (such as Declare [23], DPIL [33], or DCR Graphs [16]) since it is possible to build a 1-to-1 mapping between the SQL queries and the temporal rules that need to be checked over traces stored in a DB. Although some works have already investigated how to discover Declare rules from a DB using SQL queries [27–29], a full-fledged framework to support the event log storage in a DB and the execution of queries that can be used to support the entire spectrum of declarative process mining use cases is still missing.

In this exploratory paper, we introduce such a framework and we use it to provide a deep analysis of strengths and weaknesses of different SQL-based solutions for declarative process mining. The framework is readily available[1] for researchers and practitioners that need to analyze process data stored in relational DBs.

The paper is structured as follows. Section 2 presents the research problem. Section 3 discusses related work. Section 4 introduces the proposed framework, and discusses the DB schemas and the SQL queries supported. In Sect. 5, the framework is used to benchmark different SQL-based solutions for declarative process mining using synthetic and real-life logs. Section 6 concludes the paper and spells out directions for future work.

2 Research Problem

In this paper, we present an SQL-based framework for declarative process mining and use it to benchmark different SQL-based solutions for declarative process mining. Through the framework, we answer the following research questions:

- **RQ1**: What is the most efficient DB schema in terms of required disk space and population time?
- **RQ2**: What is the most efficient DB schema in terms of query execution time?
- **RQ3**: How does the query execution time vary for datasets with different characteristics?
- **RQ4**: How does the query execution time vary for different types of queries?

RQ1 and **RQ2** aim at understanding which one of the DB schemas existing in the literature has the highest performance in terms of population time, required disk space, and query execution time. To answer these research questions, we also test how some improvements over the existing schemas can increase

[1] https://github.com/francxx96/XEStoDB.

the DB performance. To answer **RQ3**, we show how the query execution time varies using synthetic logs with different characteristics. The query execution time measured for answering **RQ2** and **RQ3** concerns the discovery task. **RQ4** investigates, instead, the query execution time needed to run all the different types of queries provided in the proposed framework.

3 Related Work

The literature on declarative process mining covers a wide range of process mining use cases [21]. In this paper, we solve standard process mining tasks like process discovery (cf. [27,28]), conformance checking (cf. [3]), and query checking (cf. [26]), and we extend them with novel types of analysis that can be easily tackled using queries like instance-spanning process analysis [1], metric temporal rule discovery and checking [20], and local rule checking (i.e., the verification of rules in specific time intervals).

An approach for process discovery similar to the SQL-based one used in this paper is presented in [27,28]. Here, a sub-set of the standard Declare [24] templates (i.e., parameterized temporal rules) is used to define SQL queries that can be used to discover Declare models. Further investigation [29] led to the introduction of a set of queries for the discovery of Multi-Perspective Declare (MP-Declare). Other techniques (that are not based on SQL) for performing declarative process mining are available in state-of-the art process mining toolkits like RuM [2] and Declare4Py [8]. However, in order to use these tools for process analysis, the source data must be first extracted from the PAISs and then arranged in XES files.

In [30], the authors present an investigation that shows that it is possible to make the SQL queries for Declare discovery faster by using DB indexing. As mentioned in Sect. 4, the DB indexing analysis provided in [30] supports the way we designed our queries.

4 SQL-Based Declarative Process Mining Framework

In this paper, we present a full-fledged framework to perform different process mining tasks using SQL queries over relational DBs of PAISs. Figure 1 presents the conceptual overview of the framework. The framework supports two phases of the process data analysis with relational DBs.

Database Creation. A new relational DB is created following a DB schema given as input, the DB is then populated according to an input event log.

SQL-Based Declarative Process Mining. In this phase, the user chooses the process mining (PM) task to perform (i.e., discovery, query checking or conformance checking) and the query type, i.e., one of the task variants that will be introduced in Sect. 4.2. For conformance checking, the Declare model to be checked

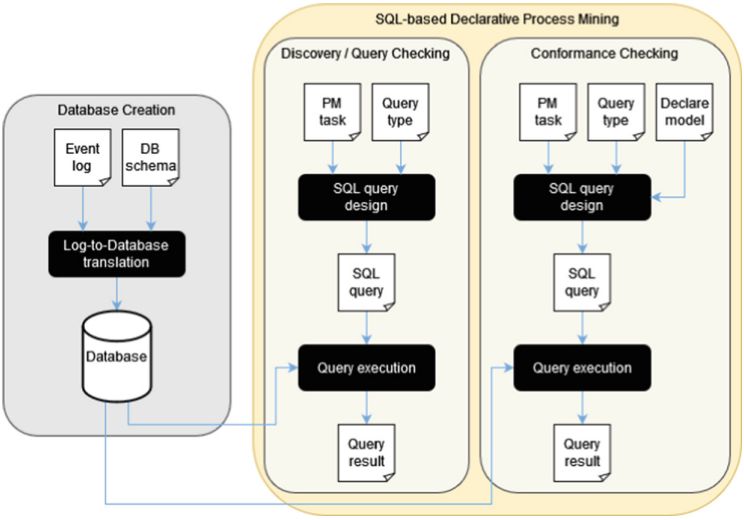

Fig. 1. Conceptual overview of the developed framework.

has to be provided by the user as well. The corresponding SQL query is designed based on the inputs. The query is then executed on the DB.

To support these two phases of the process data analysis, we implemented a wide range of queries. Some of these queries implement different declarative process mining tasks and support the *SQL-Based Declarative Process Mining* phase, others support the *Database Creation* phase using different DB schemas.

4.1 Database Creation

In the literature, different DB schemas have been proposed to store process data with the aim of minimizing both population time and disk space required to store the data. The DB schemas proposed are all fully compatible with the XES standard [32]. We selected four different relational DB schemas to be compared:

- Monolithic, composed of a single table in which each row represents an event of the log;
- DBXES, presented in [31];
- RXES, presented in [13];
- RXES+, which is an adaptation of RXES (see Fig. 2).

The design of RXES+ was intended to optimize not only the population time and the disk space needed to store the process data, but also the query execution time for process mining tasks.

Differently from DBXES and RXES, in RXES+, the *log*, *trace* and *event* tables include the mandatory XES attributes (i.e., *name*, *timestamp*, and *life-cycle transition*), so that we significantly reduce the amount of repetitions in tables $\{log|trace|event\}\_has\_attribute$ and *attribute*. Moreover, RXES is defined

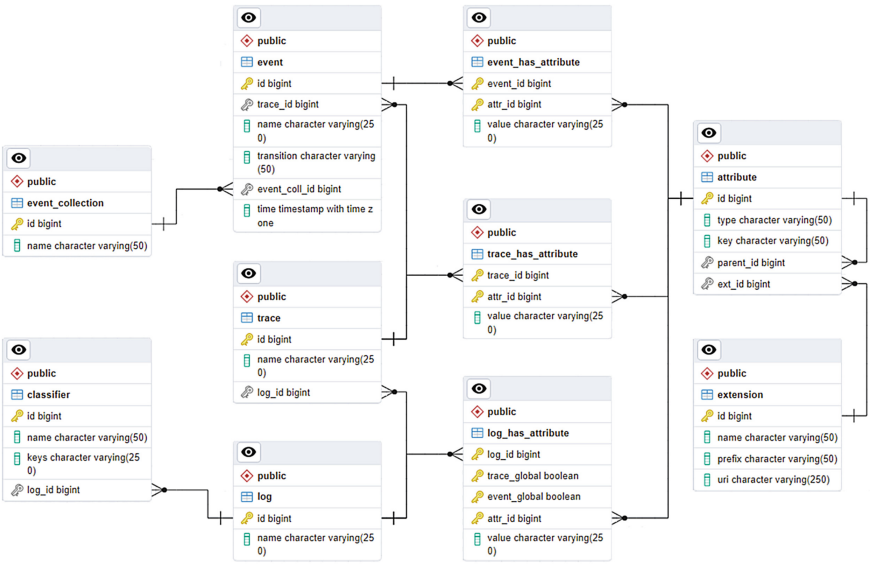

Fig. 2. RXES+ schema.

in a way that it eliminates the duplicate traces/events inside a log, i.e., when populating the DB, if an event or a trace is exactly the same as another event or trace already present in the DB, this element is not repeated but linked to the existing one. However, checking the existence of an event or a trace in the DB is an unnecessarily heavy task. Indeed, from an analysis of the logs existing in the literature (see Table 1), we can see that, since events are always provided with a timestamp, the occurrence of duplicate traces/events is extremely rare (this only happens if multiple events with exactly the same attributes occur exactly at the same time). For this reason, RXES+ has been designed to allow duplicates. As we can see from our experiments, this does not significantly affect the DB size and the query execution time but dramatically reduces the population time.

4.2 SQL-Based Declarative Process Mining

SQL queries are a powerful instrument for implementing in a straightforward way a large range of analysis types over process data recorded in a DB. Here, we present the query types available in our framework. All the queries use a temporary table *@event*, which allows the queries to be formulated exactly in the same way independently of the schema. The temporary table *@event* is built with a different query depending on the schema. For example, for the RXES+ schema *@event* is built with the query:

```
1 DECLARE @event TABLE ( log_id BIGINT, trace_id BIGINT,
      task VARCHAR(300), [timestamp] DATETIME2(3) )
2
3 INSERT @event
```

```
SELECT t.log_id, e.trace_id, e.name + '_' + e.
    transition,
    e.[timestamp]
FROM trace t JOIN event e ON t.id = e.trace_id;
```

Discovery. For process discovery, i.e., for the identification of a set of rules (based on a Declare template specified by the user) satisfied with a minimum support[2] in an input log, we follow the approach introduced in [27,28], where, first, the input template is instantiated into different candidate constraints (obtained by replacing the template parameters with all the possible combinations of activities available in the log), and, then, the candidate constraints are checked to compute their support.

In particular, for each SQL query defined from now on, the *Support* is computed as follows:

```
CAST(COUNT(*) AS FLOAT) / CAST( (SELECT COUNT(*) FROM
    @event WHERE task='TaskA') AS FLOAT)
```

Starting from the queries presented in [27,28] - the Baseline (BS) query set - developed for the discovery of standard Declare rules, we designed a new query set - the Join query set - in order to improve the query execution time for discovery. In particular, the new query set was designed with the aim of reducing possible performance bottlenecks; in order to achieve this, we used the query plan [14] produced when executing each query. From the plans, we noticed that, before running any queries, the DBMS always sorts the events in the DB when the queries contain explicit JOIN statements. This conclusion is similar to the one found in [30], where a systematic DB indexing analysis is conducted. Therefore, we re-designed the queries to benefit of this automatic DB indexing executed by the DBMS.

Example 1. In Declare, the *response* template instantiated with activation A and target B indicates that, when activity A is executed, it must be eventually followed by B. The discovery from an input log of rules of type *response* can be obtained with the following query relying on explicit JOIN statements:

```
SELECT 'Response', TaskA, TaskB, Support
FROM (
    SELECT a.trace_id, a.task AS TaskA, b.task AS TaskB
    FROM @event a JOIN @event b ON (
        a.log_id = b.log_id
        AND a.trace_id = b.trace_id
        AND a.task != b.task
        AND a.[timestamp] < b.[timestamp]
    ) GROUP BY a.trace_id, a.task, a.[timestamp], b.
        task
) subquery
GROUP BY TaskA, TaskB;
```

[2] Here the support corresponds to the *event support* introduced in [7].

Conformance Checking. For conformance checking, we follow the approach introduced in [3]. The input here is not a generic template but a Declare model, i.e., a set of concrete rules, which are instantiations of templates with real activities. The outcome is the support in the log of each rule in the model.

Example 2. The conformance checking of a rule of type *response* instantiated over activities *Receive Payment* and *Send Receipt* wrt. an input log is obtained from the one seen in Ex1 by changing line 7 as follows:

```
7        AND a.task='Receive Payment' AND b.task='Send
             Receipt'
```

Query Checking. This type of analysis was first introduced in [26]. The input here is a partial instantiation of a template, i.e., a template where only one of the parameters is replaced with a real activity, while the other one remains unspecified. In addition, a minimum support is also specified. The outcome is the discovery from an input log of rules of the specified format, and satisfied in the log with the specified minimum support.

Example 3. The query checking of a rule of type *response* instantiated with activation *Receive Payment* and target left unspecified is obtained from the one seen in Example 1 by changing line 7 as follows:

```
7        AND a.task='Receive Payment' AND b.task!=a.task
```

Additional types of analysis The standard approaches for discovery, conformance, and query checking just introduced can be extended using variants of the standard queries, which provide facilities to solve well-known problems in declarative process mining, such as:

- Instance-Spanning analysis (IS) [1], which considers the whole log as a single trace obtained by ordering the events by timestamp;
- Local Rule Checking (LRC), which checks the validity of a rule only within a given time interval;
- Metric Temporal rule discovery (MT) [20], which enriches the query for the discovery of standard Declare rules with information about the minimum/average/maximum temporal distance between activation and target activities;
- Validity Intervals analysis (VAL), which finds the time intervals in a trace in which a Declare rule is valid;
- Attribute Range (RNG) analysis, which finds for a given attribute the range of values it gets in a given time interval.

These variants can also be easily combined together to build custom queries that are useful for a particular need of the end user, e.g., it is possible to combine IS with LRC in order to have an instance-spanning query restricted to a given time interval.

Example 4. The following query implements the Instance-Spanning discovery of *response* rules:

```
SELECT 'Response', TaskA, TaskB, Support
FROM (
    SELECT a.trace_id, a.task AS TaskA, b.task AS TaskB
    FROM @event a JOIN @event b ON (
        a.log_id = b.log_id
        AND a.task != b.task
        AND a.[timestamp] < b.[timestamp]
    ) GROUP BY a.trace_id, a.task, a.[timestamp], b.
        task
) subquery
GROUP BY TaskA, TaskB;
```

The following query implements the Local Rule Checking of the *response* template instantiated over activities *Receive Payment* and *Send Receipt* in the time interval spanning from 2020-01-01 00:00:00.000 to 2021-12-31 23:59:59.999:

```
DECLARE @interval_start DATETIME2(3)='2020-01-01 
    00:00:00.000',
    @interval_end DATETIME2(3)='2021-12-31 23:59:59.999
        ';

SELECT 'Response', TaskA, TaskB, Support
FROM (
    SELECT a.trace_id, a.task AS TaskA, b.task AS TaskB
    FROM @event a JOIN @event b ON (
        a.log_id = b.log_id AND a.trace_id = b.trace_id
        AND a.task = 'Receive Payment' AND b.task = '
            Send Receipt'
        AND a.[timestamp] < b.[timestamp]
    ) WHERE a.[timestamp] >= @interval_start
        AND a.[timestamp] < @interval_end
        AND b.[timestamp] >= @interval_start
        AND b.[timestamp] < @interval_end
    GROUP BY a.trace_id, a.task, a.[timestamp], b.task
) subquery
GROUP BY TaskA, TaskB;
```

The following query implements the Metric Temporal discovery of *response* rules:

```
SELECT 'Response', TaskA, TaskB, Support,
    MIN(TD) AS min_TD, AVG(TD) AS avg_TD, MAX(TD) AS
        max_TD
FROM (
    SELECT a.trace_id, a.task AS TaskA, b.task AS TaskB
        ,
        MIN(DATEDIFF(SECOND, a.[timestamp], b.[
            timestamp])) AS TD
    FROM @event a JOIN @event b ON (
```

```
7           a.log_id = b.log_id AND a.trace_id = b.trace_id
8           AND a.task != b.task AND a.[timestamp] < b.[
                timestamp]
9       ) GROUP BY a.trace_id, a.task, a.[timestamp], b.
            task
10  ) subquery
11  GROUP BY TaskA, TaskB;
```

For space limitations, we do not report here the queries returning the Validity Intervals of a rule and the Attribute Range of an attribute. The interested reader can find the details about these queries at https://github.com/francxx96/XEStoDB.

5 Benchmarks

To answer the research questions introduced in Sect. 2, we performed experiments on synthetic and real-life logs. In the following sections, we first describe the experimental setting, i.e., describe the logs and metrics used in the experimentation, then, we discuss the experimental results to answer the research questions.

5.1 Experimental Setting

As already mentioned, we validated our SQL-based framework by considering both synthetic and real-life logs. With the real-life logs, we wanted to demonstrate the applicability of the framework to well-known benchmarks in the process mining field. In particular, we considered six logs, most of them presented in past editions of the Business Process Intelligence Challenge (BPIC):

- SEPSIS, recording the treatment of incoming patients with sepsis in a hospital [22];
- ROAD, related to a road traffic fines management process [18];
- FINANC, pertaining to a loan application process (provided for the BPIC 2012) [9];
- LOAN, a richer version of FINANC (provided for the BPIC 2017) [10].
- REIMB, pertaining to a reimbursement process for international declarations (provided for the BPIC 2020) [11];
- TRAVEL, related to the management of travel permits (provided for the BPIC 2020) [12];

Table 1 reports the characteristics of the real-life logs. These logs are widely heterogeneous ranging from simple to very complex, with a log size ranging from 1,050 traces (for the SEPSIS log) to 150,370 traces (for the ROAD log). A similar variety can be observed in the number of event classes (i.e., activities executed in the log), ranging from 11 to 51. Moreover, the trace length also varies from very short traces (containing only two events), to very long traces (containing 185 events). The table also shows the percentages of duplicate events in each log. In this respect, we can see that, except for the ROAD log, the percentage of duplicate events is always equal to zero or very close to it.

Table 1. Descriptive statistics of real-life logs.

Log name	Total traces	Total events	Event classes	Duplicate events	Trace length		
					min	avg	max
SEPSIS	1,050	15,214	16	<0.01%	3	14	185
ROAD	150,370	561,470	11	79.63%	2	4	20
FINANC	13,087	262,200	36	<0.01%	3	20	175
LOAN	31,509	1,160,405	26	0%	9	37	177
REIMB	6,449	72,151	34	0.04%	3	11	27
TRAVEL	7,065	86,581	51	0%	3	12	90

Synthetic logs were created using the *ASP log generator* [6] implemented in the declarative process mining tool RuM [2]. They are intended to prove the scalability of the presented framework wrt. logs with specific characteristics (i.e., number of distinct event classes, number of traces in the log, number of events in a trace) in a controlled environment. We built several different synthetic logs, each named using the format `clsXXXtrcXXXevtXXX`. For example, `cls10trc100evt30` identifies a log containing 10 different event classes and 100 traces, each including 30 events.

The performance metrics we considered in our experimentation to answer the research questions are:

- Required disk space to store a log in a DB, which is a measure of the DB redundancy degree;
- DB population time;
- Event insertion time, which measures the time needed for inserting an event in the DB (when measured for subsequent insertions, it might happen that the insertion time is higher for events inserted later in the DB);
- Query execution time.

All the experimental material can be found in the repository available at https://github.com/francxx96/XEStoDB, which contains:

- Translation scripts (Java 11) from XES-formatted files to each type of DB considered;
- SQL schemas reproducing the (empty) DBs;
- SQL dumps of the DBs already populated with the datasets used in our experiments;
- SQL queries for implementing all the declarative process mining task discussed in this paper.

We performed our experiments on a machine with an Intel Xeon E5-2690 CPU (dual core, 2.60 GHz), Windows Server 2019 OS and 16 GB RAM. The DBMS we used to define DB schemas and queries is Microsoft SQL Server 2019.

5.2 Results

RQ1. What Is the Most Efficient DB Schema in Terms of Required Disk Space and Population Time? To answer this research question, we created, for each considered real-life log, four DBs (one for each considered schema)

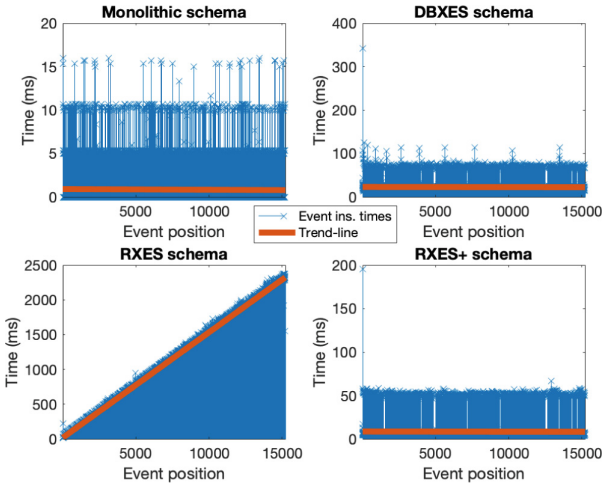

Fig. 3. Subsequent event insertion times for SEPSIS.

Table 2. Disk space for log storage and average population time.

DB schema	Disk space (MB)						Population time (s)					
	SEPSIS	ROAD	FINANC	LOAN	REIMB	TRAVEL	SEPSIS	ROAD	FINANC	LOAN	REIMB	TRAVEL
Monolithic	8.50	184.98	110.03	627.00	53.24	91.78	34.69	1,211.73	560.31	2,795.66	156.03	216.90
DBXES	7.63	180.30	73.93	–	43.55	49.13	364.09	15,551.52	14,440.95	–	6,034.75	7,303.98
RXES	8.50	–	–	–	–	–	18,060.88	–	–	–	–	–
RXES+	5.08	92.59	30.19	318.96	21.91	26.30	134.89	5,063.72	1,203.21	13,406.82	896.47	1,085.52

containing the process data of that log. A comparison in terms of required disk space needed to store the logs using the different DB schemas and their population times[3] (averaged over 5 runs) can be seen in Table 2. Here, we can observe that, as expected, the Monolithic schema has the highest degree of redundancy and occupies in all cases the highest amount of disk space (in some cases more than three times wrt. RXES+). This is a critical issue for large real-life logs. In addition, the large amount of disk space occupied, forces the Monolithic schema to have an upper-bound on the number of attributes that can be stored in the DB given by the limited number of columns admitted by the DBMSs for a single table. The RXES+ schema, instead, uses always less space than other schemas for all the analyzed logs. For what concerns the population time, the Monolithic schema is the fastest one. This is due to its simple structure (it is composed of a single table containing all the log data) that does not require to update DB-related constraints (e.g., foreign keys) when inserting a new event. The population time of RXES is extremely high since its structure requires to check the presence of duplicate events/traces at each insertion. This is confirmed by the more detailed analysis conducted on the SEPSIS log shown in Fig. 6 3. The plots in the figure indicate the time needed for subsequent event insertions

[3] We set a timeout on the population scripts and each script that did not end within 24 h was stopped. Dashes in the tables mean that the corresponding scripts reached the timeout.

Table 3. Query execution time comparison between the query sets.

DB schema	Average query time (s)													
	Response		Alternate Response		Chain Response		Precedence		Alternate Precedence		Chain Precedence		Responded Existence	
	BS	Join	BS	Join	BS	Join	BS	Join	BS	Join	BS	Join	BS	Join
SEPSIS														
Monolithic	0.585	0.506	1.111	0.587	1.215	0.427	0.564	0.543	1.405	0.538	1.595	0.429	1.193	0.975
DBXES	1.176	1.239	1.854	1.259	1.999	1.037	1.319	1.299	2.140	1.208	2.382	1.050	1.949	1.714
RXES	0.676	0.636	1.213	0.720	1.348	0.507	0.698	0.675	1.522	0.649	1.735	0.502	1.318	1.107
RXES+	0.575	0.524	1.105	0.602	1.225	0.452	0.561	0.547	1.394	0.542	1.575	0.447	1.173	0.990
ROAD														
Monolithic	9.40	8.04	25.78	8.31	31.43	4.91	9.06	7.86	25.97	8.06	31.87	4.76	22.78	14.55
DBXES	13.73	12.56	31.08	13.28	36.39	9.85	13.98	12.77	31.39	12.92	37.03	9.75	27.63	19.73
RXES+	9.44	8.09	26.43	8.44	31.74	5.02	9.28	7.90	26.53	8.12	32.44	4.89	22.93	14.57
FINANC														
Monolithic	36.72	32.57	39.99	22.17	57.59	15.39	36.26	33.58	44.83	18.54	74.54	15.56	60.86	68.38
DBXES	46.06	41.63	50.54	31.16	67.62	22.62	45.93	43.08	55.67	27.74	84.30	23.08	70.57	79.73
RXES+	37.25	32.41	40.87	21.75	57.21	15.65	36.45	33.51	45.66	18.41	74.64	16.20	60.13	68.09
LOAN														
Monolithic	542.76	202.87	355.58	166.87	566.16	93.29	533.45	207.00	369.44	143.76	663.95	96.67	843.42	424.74
RXES+	540.37	204.00	350.52	167.32	568.96	95.01	546.44	207.66	374.10	145.01	678.89	97.44	895.67	422.88
REIMB														
Monolithic	8.79	1.65	9.20	1.91	13.44	1.09	8.94	1.62	9.26	1.86	13.98	1.11	16.43	2.71
DBXES	13.20	5.49	14.94	6.38	20.42	4.69	13.47	6.18	15.15	6.50	20.77	4.75	22.49	8.60
RXES+	9.77	2.72	11.67	2.98	15.74	1.30	10.18	2.64	11.79	2.92	16.19	1.28	19.14	4.79
TRAVEL														
Monolithic	22.63	4.22	20.95	4.36	27.72	2.02	22.59	4.09	20.62	4.23	28.03	2.04	32.85	7.45
DBXES	25.60	7.79	24.57	8.63	32.10	6.09	26.75	8.41	24.72	8.64	33.15	6.20	37.34	12.02
RXES+	22.28	4.18	20.35	4.35	27.37	1.98	22.54	4.07	20.67	4.21	27.98	1.99	33.13	7.43

for the four considered schemas. The time required for subsequent event insertions for RXES grows linearly with the number of events already inserted, while it remains constant for the other schemas. RXES+ is faster than both DBXES and RXES. Overall, RXES+ guarantees a good trade-off between the disk space needed to store the process data and the population time.

RQ2. What Is the Most Efficient DB Schema in Terms of Query Execution Time? To answer this research question, we executed all the queries in the two discovery query sets BS and Join for all the considered real-life logs stored using all the considered DB schemas. Table 3 shows the query execution times measured by executing the BS and the Join query set. The gray background in the table indicates the schema that performed best on the same log and the same query. The query execution times displayed in the tables are averaged over 5 runs. The results highlight that the Monolithic and RXES+ schemas achieve very similar performance on both query sets, and perform better than DBXES and RXES. Almost all the queries in the Join set run faster than the baseline queries.

RQ3. How Does the Query Execution Time Vary for Datasets with Different Characteristics? To answer this research question, we performed a set of controlled experiments using synthetic logs generated by varying the number of event classes, the number of traces, and the number of events in each

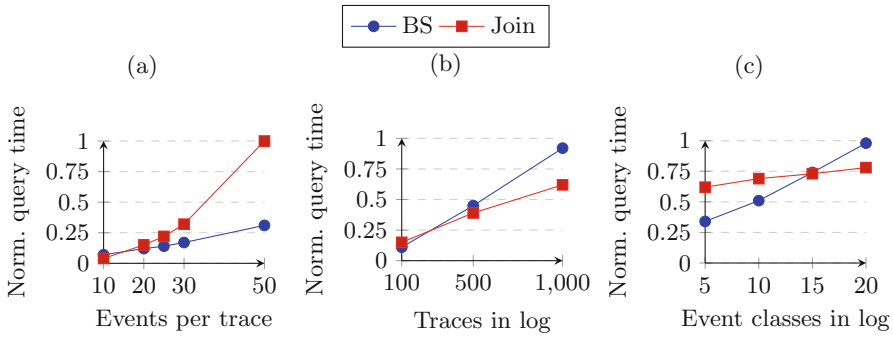

Fig. 4. Query execution time for Discovery task wrt. (4a) number of events per trace, (4b) number of traces in the log, (4c) number of event classes in the log.

trace. In particular, we generated three sets of synthetic logs; each set fixes two of the above parameters, while changing the remaining one.

For testing how the trace size affects the query execution time, we generated five synthetic logs containing 10 event classes and 100 traces of size 10, 20, 25, 30, and 50, respectively. For testing how the log size affects the query execution time, we generated three synthetic logs containing 10 event classes, traces of size 20 and with log size equal to 100, 500, and 1000, respectively. Finally, for testing how a different number of event classes affects the query execution time, we generated four synthetic logs with log and trace size equal to 100 and 20, respectively, and containing 5, 10, 15, and 20 event classes. The query execution time was, again, measured running all the queries in the two discovery query sets BS and Join and was normalized over the results obtained for all the queries in the two query sets, in order to have a single performance indicator for all queries in each set. We used the RXES+ schema for these experiments.

The results are shown in Fig. 4. In Fig. 4a, we can observe that the Join query set is more efficient when traces are shorter, while grows exponentially as the traces become longer. This is due to the structure of the Join queries, which leverages DB indexing (improving the performance for traces with less than 20 events) and a Cartesian product, containing all the possible combinations of the events in a trace, which becomes larger when the trace size increases. Figure 4b shows, instead, that the Join query set scales better than BS as the log size grows, and Fig. 4c shows how the query execution time for both BS and Join grows linearly when the number of event classes increases, with Join becoming more efficient for logs containing more than 15 event classes.

To sum up, for logs with more than 15 event classes and more than 500 traces each containing less than 20 events (which are common characteristics for many real-life logs) the queries in the Join set are more efficient. However, for traces particularly long, it is better to use queries that do not rely on explicit JOIN statements.

Table 4. Query execution time for each query type.

	Average query time (s)						
Query type	Response	Alternate Response	Chain Response	Precedence	Alternate Precedence	Chain Precedence	Responded Existence
SEPSIS							
STD	0.52	0.60	0.45	0.55	0.54	0.45	0.99
IS	471.55	491.96	820.21	605.46	473.83	814.45	0.95
LRC	0.65	0.86	0.74	0.65	0.83	0.76	1.03
MT	0.84	0.84	0.62	0.83	0.77	0.61	0.82
VAL	0.96	1.94	4.16	1.23	6.22	3.48	2.68
ROAD							
STD	8.09	8.44	5.02	7.90	8.12	4.89	14.57
IS	–	–	–	–	–	–	42.69
LRC	7.00	8.94	5.32	6.55	8.68	5.19	11.37
MT	0.84	8.71	5.19	8.25	8.47	5.07	8.40
VAL	0.96	19.66	11.99	11.51	20.02	12.12	26.72
FINANC							
STD	32.41	21.75	15.65	33.51	18.41	16.20	68.09
IS	–	–	–	–	–	–	65.38
LRC	22.13	23.97	16.52	23.32	20.67	17.18	41.76
MT	37.91	23.82	17.38	38.64	20.15	18.09	38.02
VAL	42.84	69.79	105.42	51.92	90.67	109.70	102.22
LOAN							
STD	204.00	167.32	95.01	207.66	145.01	97.44	422.88
IS	–	–	–	–	–	–	677.36
LRC	230.16	193.64	97.39	233.38	166.22	100.56	475.61
MT	242.90	181.59	104.70	249.01	157.58	106.87	249.84
VAL	322.73	434.47	636.92	363.25	482.77	658.32	1,032.00
REIMB							
STD	2.72	2.98	1.30	2.64	2.92	1.28	4.79
IS	–	–	–	–	–	–	15.49
LRC	3.08	3.77	2.09	3.12	3.82	2.21	5.44
MT	3.38	3.59	1.39	3.33	3.57	1.40	3.49
VAL	5.46	7.57	5.63	5.39	7.54	6.27	12.44
TRAVEL							
STD	4.18	4.35	1.98	4.07	4.21	1.99	7.43
IS	–	–	–	–	–	–	29.88
LRC	4.68	5.38	3.05	4.49	5.39	3.24	6.72
MT	5.08	5.16	2.13	4.91	4.90	2.16	4.99
VAL	7.64	10.76	9.15	7.86	11.11	10.44	17.25

Table 5. Query execution time for RNG query type.

Query type	Average query time (s)					
	SEPSIS	ROAD	FINANC	LOAN	REIMB	TRAVEL
RNG	0.14	0.18	0.09	14.25	1.23	0.70

RQ4. How Does the Query Execution Time Vary for Different Types of Queries? To answer this research question, we executed all the additional types of queries defined in Sect. 4 using RXES+ as schema on each log.[4]

Table 4 compares, for each Declare template, the execution time (averaged over 5 runs) of the standard queries in the Join query set (STD) with the IS, LRC, MT, VAL. Here, IS, LRC, and MT are standard discovery tasks, while VAL returns the validity intervals of all the Declare rules obtained by instantiating each template with all the combinations of event classes available in the log. Table 5 shows results for the RNG query execution time. RNG returns all the value ranges of all the attributes available in each log in a fixed time interval. The details about the queries used in this experiment can be found at https://github.com/francxx96/XEStoDB.

We have already seen from the experiments on the synthetic logs that the execution time of the queries in the Join set grows exponentially as the trace size grows. Since the instance-spanning queries consider the whole log as a single trace, the IS queries require much more time to be executed. To solve this issue, in our repository, we provide also for this type of queries a version that does not use explicit JOIN statements.

6 Conclusion

In this paper, we proposed an SQL-based framework for declarative process mining. We proposed different queries to support the three main use cases of declarative process mining, i.e., process discovery, conformance checking, and query checking. We also presented an extensive cross-benchmark comparison we conducted using several synthetic and real-life logs for investigating the performance of different DB schemas and different types of queries.

The evaluation has been conducted using Microsoft SQL Server 2019 as DBMS. Nonetheless, the conclusions drawn are valid in general. In particular, the improved performance of RXES+ in terms of disk space needed to store a log and the insights derived from the experiments on the DB population time are clearly valid independently of the DBMS used. In addition, the improvements in the query execution time obtained with the use of explicit JOIN statements are also valid in general provided that the queries are executed on an indexed DB. Another observation that is worth mentioning is that all the queries presented in this paper are easily applicable to any (proprietary) DB schema with the

[4] We set a timeout of 30 min on the query scripts.

only requirement that the DB contains timestamped events somehow grouped together (into traces), which is the most basic requirement needed to conduct any type of process mining analysis on a dataset (only the query for building the temporary table *@event* must be rewritten when using a new DB schema). Portability is, in general, a significant advantage of the proposed framework and this is the reason why we developed it by relying only on standard SQL clauses. Although other more sophisticated SQL clauses could be used to improve the overall performance of the framework, these solutions could affect its portability across different DBMSs.

We think that the investigations conducted in this paper can be considered as an important basis for researchers who want to develop techniques for process mining based on SQL since they give several insights about the main bottlenecks and possible issues that can come up when DBs are used for process analytics. This work can be, in the future, extended towards several directions. First, a systematic comparison with the techniques for performing declarative process mining available in toolkits like RuM [2] and Declare4Py [8] could be conducted.

Even if we defined basic queries over the data attributes attached to events in a log (the RNG queries), more sophisticated queries could be defined, for example, for checking MP-Declare rules. These queries might represent the basis for novel approaches for discovery, conformance checking and query checking based on MP-Declare. Also, it would be easy to compute, using SQL queries, metrics for measuring the "interestingness" of a Declare rule [5] that go beyond the support we use in this paper (e.g., confidence).

The use of different query languages like PQL [25] could be investigated in the context of declarative process mining. Smarter strategies for storing event data in DBs like the ones investigated in [7] could help improving the query execution time. Other DB schema can be built for other standards for storing process information in event logs like the recent object-centric standards, such as XOC [19] and OCEL [15].

Another avenue for future work is the development of conformance checking SQL queries providing richer feedback to the user like trace alignments. Finally, the use of more advanced instruments from DB theory, such as the use of temporal DBs [17], could be investigated with the aim of improving the performance of the proposed framework.

References

1. Aamer, H., Montali, M., Van den Bussche, J.: What Can Database Query Processing Do for Instance-Spanning Constraints? CoRR abs/2206.00140 (2022)
2. Alman, A., Di Ciccio, C., Haas, D., Maggi, F.M., Nolte, A.: Rule Mining with RuM. In: 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, 4–9 October 2020, pp. 121–128 (2020)
3. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)

4. Calì, A., Calvanese, D., Lenzerini, M.: Data integration under integrity constraints. In: Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, pp. 335–352. Springer (2013). https://doi.org/10.1007/978-3-642-36926-1
5. Cecconi, A., De Giacomo, G., Di Ciccio, C., Maggi, F.M., Mendling, J.: Measuring the interestingness of temporal logic behavioral specifications in process mining. Inf. Syst. **107**, 101920 (2022)
6. Chiariello, F., Maggi, F.M., Patrizi, F.: ASP-Based Declarative Process Mining. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI) (2022)
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., Riva, F., Maggi, F.M., Shikhizada, A.: Declare4Py: a python library for declarative process mining. In: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2022, pp. 117–121 (2022)
9. van Dongen, B.F.: BPI Challenge 2012 (Apr 2012)
10. van Dongen, B.F.: BPI Challenge 2017 (Feb 2017)
11. van Dongen, B.F.: International Declarations Log. BPI Challenge 2020 (Mar 2020)
12. van Dongen, B.F.: Travel Permits Log. BPI Challenge 2020 (Mar 2020)
13. van Dongen, B.F., Shabani, S.: Relational XES: data management for process mining. In: CAiSE Forum 2015, pp. 169–176 (2015)
14. Ewen, S., Kache, H., Markl, V., Raman, V.: Progressive query optimization for federated queries. In: Advances in Database Technology - EDBT 2006. vol. 3896, pp. 847–864 (2006)
15. Ghahfarokhi, A., Park, G., Berti, A., van der Aalst, W.M.P.: OCEL: A Standard for Object-Centric Event Logs, pp. 169–175 (07 2021)
16. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed Dynamic Condition Response Graphs. J. Log. Algebr. Program. **82**(5–7), 164–185 (2013)
17. Kulkarni, K.G., Michels, J.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
18. de Leoni, M., Mannhardt, F.: Road Traffic Fine Management Process (Feb 2015)
19. Li, G., de Murillas, E.G.L., de Carvalho, R.M., van der Aalst, W.M.P.: Extracting object-centric event logs to support process mining on databases. In: Information Systems in the Big Data Era, pp. 182–199 (2018)
20. Maggi, F.M.: Discovering metric temporal business constraints from event logs. In: Johansson, B., Andersson, B., Holmberg, N. (eds.) BIR 2014. LNBIP, vol. 194, pp. 261–275. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11370-8_19
21. Maggi, F.M.: Declarative Process Mining. In: Sakr, S., Zomaya, A.Y. (eds) Encyclopedia of Big Data Technologies. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-77525-8_92
22. Mannhardt, F.: Sepsis Cases - Event Log (Dec 2016)
23. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: IEEE International EDOC Conference 2007, pp. 287–300 (2007)
24. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), pp. 287–300 (2007)
25. Polyvyanyy, A., ter Hofstede, A.H.M., La Rosa, M., Ouyang, C., Pika, A.: Process query language: Design, implementation, and evaluation. CoRR abs/1909.09543 (2019)

26. Räim, M., Di Ciccio, C., Maggi, F.M., Mecella, M., Mendling, J.: Log-based understanding of business processes through temporal logic query checking. In: Meersman, R. (ed.) OTM 2014. LNCS, vol. 8841, pp. 75–92. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45563-0_5
27. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and Customisable declarative process mining with SQL. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 290–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39696-5_18
28. Schönig, S.: SQL Queries for Declarative Process Mining on Event Logs of Relational Databases. CoRR abs/1512.00196 (2015)
29. Schönig, S., Di Ciccio, C., Maggi, F.M., Mendling, J.: Discovery of multi-perspective declarative process models. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 87–103. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_6
30. Schönig, S., Di Ciccio, C., Mendling, J.: Configuring SQL-based process mining for performance and storage optimisation. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 94–97. SAC 2019 (2019)
31. Syamsiyah, A., van Dongen, B.F., van der Aalst, W.M.P.: DB-XES: enabling process discovery in the large. In: Ceravolo, P., Guetl, C., Rinderle-Ma, S. (eds.) SIMPDA 2016. LNBIP, vol. 307, pp. 53–77. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74161-1_4
32. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Information Systems Evolution - CAiSE Forum. vol. 72, pp. 60–75 (2010)
33. Zeising, M., Schönig, S., Jablonski, S.: Towards a Common Platform for the Support of Routine and Agile Business Processes. In: Collaborative Computing: Networking, Applications and Worksharing (2014)