What Can Database Query Processing Do for Instance-Spanning Constraints?

Heba Aamer[1][0000−0003−0460−8534], Marco Montali[2][0000−0002−8021−3430], and Jan Van den Bussche[1][0000−0003−0072−3252]

[1] Hasselt University, Belgium
{heba.mohamed,jan.vandenbussche}@uhasselt.be
[2] Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract. In the last decade, the term *instance-spanning constraint* has been introduced in the process mining field to refer to constraints that span multiple process instances of one or several processes. Of particular relevance, in this setting, is checking whether process executions comply with constraints of interest, which at runtime calls for suitable monitoring techniques. Even though event data are often stored in some sort of database, there is a lack of database-oriented approaches to tackle compliance checking and monitoring of (instance-spanning) constraints. In this paper, we fill this gap by showing how well-established technology from database query processing can be effectively used for this purpose. We propose to define an instance-spanning constraint through an ensemble of *four database queries* that retrieve the satisfying, violating, satisfying-pending, and violating-pending cases of the constraint. In this context, the problem of compliance monitoring then becomes an application of techniques for incremental view maintenance, which is well-developed in database query processing. In this paper, we argue for our approach in detail, and, as a proof of concept, present an experimental validation using the DBToaster incremental database query engine.

Keywords: Compliance monitoring · SQL · Databases.

Q: What's in a constraint?
A: Two (or four) database queries!

1 Introduction

Constraints that are posed over business processes can be very general and can refer to a variety of requirements [26]. Non-compliance of certain constraints can be very costly and risky, so compliance checking and monitoring are of utmost importance to the enterprise [36]. The following are various examples of such constraints. We will refer to these again in the remainder of this paper.

C1 *"A shipping car can be used to deliver at most seven packages per day"*
C2 *"The time taken to deliver a package is between two and five days"*

C3 *"Conducting a patient's surgery must be preceded by examining the patient"*
C4 *"Packages that are delivered to the same neighbourhood on the same day must be delivered by the same shipping car"*

Constraints can be very simple in terms of their scope, i.e., the process instances they involve, and the conditions they impose such as C2 and C3. These are examples of constraints to be enforced on activity instances belonging to the same process instance. This type of constraint is often referred to as *intra-instance* [35,36]. On the other hand, there are constraints that can be much more complex, both in their scope and in the conditions they impose. Specifically, constraints where the scope spans multiple process instances, or combinations of entities involved in multiple process instance, have been referred to as *inter-instance* [35,29], or, more recently, *instance-spanning* constraints (ISCs) [15,32]. C1 and C4 are examples of ISCs.

Although our focus is on studying ISCs, similar complex features would be required when checking intra-instance constraints on complex processes. In what follows, we will hence just talk about (process) *constraints* in general.

Constraints must be checked against execution logs, which are files or databases holding data about past and current executions of all process instances in the enterprise. *Post-mortem checking* and *compliance monitoring* are the two types of compliance checking that are commonly distinguished depending on the nature of the data used. The latter is more challenging since it checks the execution of the currently running process instances, for a live log.

There is a striking similarity between the problem of compliance monitoring and the problem of *incremental view maintenance*, a well-researched problem in databases [20,19,18,9,23,24]. There, a *view* is the materialized result of a (possibly complex) query posed against a database. The problem of view maintenance is then to keep the view consistent with its definition under changes to the database. In general, these changes may be CRUD operations such as in particular insertions, deletions, or updates. This is perfectly in line with the execution of a process, where events witness the execution of tasks that, in turn, are typically associated to CRUD operations used to persist relevant event data in an underlying storage.

In this paper, we put forward the idea that incremental view maintenance is applicable to do compliance monitoring. We need to answer three questions: (1) What is the database? (2) What are the updates? (3) What is the query?

The first two questions are easily answered: the database is a representation of the contents of the log, and events trigger insertions to the log to leave a trace about their occurrence. In this context, only insertion operations are thus used, to append the occurrence of an event to those occurred before. Every insertion, triggered by the execution of some activity instance, stores the corresponding event data in the database, including the timestamp of the event and which data payload it carries.

What is then the query? To answer this question, we first need to indicate which dimensions we want to tackle when expressing constraints. Given the nature of ISCs, we want to comprehensively tackle multi-perspective constraints

dealing with several cases and their control-flow, time, and data dimensions. Instead of defining a specific constraint language that can accommodate such different perspectives, we directly employ full-fledged SQL for the purpose. Hence, a constraint is expressed as a query or, more precisely, an ensemble of queries, the number of which depends on whether compliance has to be assessed post-mortem or at runtime. In post-mortem checking, a constraint is expressed as a pair of queries $(Q_{\text{case}}, Q_{\text{viol}})$, which will be defined in Definition 1. Intuitively, Q_{case} defines the "scope" of the constraint, while Q_{viol} defines the violating subset of Q_{case}.

At runtime, we take inspiration from previous works in monitoring processes and temporal logic specifications [5,27,11], and consider that each constraint may be, in principle, in one of four possible states: currently satisfied (resp., currently violated), that is, satisfied (resp., violated) by the current event data, but with a possible evolution of the system that will lead to violation (resp., satisfaction); permanently satisfied (resp., permanently violated), that is, satisfied (resp., violated) by the current event data, and staying in that state no matter which further events will occur in the future. For well-studied languages only tackling the control-flow dimension, such as variants of linear temporal logics over finite traces, such states can all be automatically characterized starting from a single formula formalizing the constraint of interest [12]. This is not the case for richer languages tackling also the data dimension, as in this setting reasoning on future continuations is in general undecidable [13,7]. We therefore opt for a pragmatic approach where constraint states are manually identified by the user through dedicated queries [29,8]. In particular, a monitored constraint comes with an ensemble of four queries $(Q_{\text{case}}, Q_{\text{viol-perm}}, Q_{\text{viol-pend}}, Q_{\text{sat-pend}})$, which will be defined later in Definition 2.

To monitor constraints, we have used the system DBToaster for incremental query processing [23,24,34] in a proof-of-concept experiment. We monitor a number of realistic constraints on experimental data taken from the work by Winter et al. [36]. We demonstrate our approach in Sections 2 and 3 of the paper.

Importantly, while we employ here the de-facto standard query language in databases, SQL, any other general data model (capable of suitably representing execution logs) with a sufficiently expressive declarative query language would do as well. Examples are the RDF data model with SPARQL, or graph databases with Cypher, which have been recently used in the context of object-centric process mining [14]. It should be noted, however, that incremental query processing is the most advanced for SQL. Indeed, relational database management systems are still the most mature database technology in development since the 1970s.

The paper is organized as follows. In Section 2, we formalize our approach, discuss examples of constraints and express them as SQL queries. In Section 3, we elaborate on the problem of compliance monitoring. In Section 4, we present the experimental results. In Section 5, we discuss query language extensions for sequences that can be useful as an approach. We conclude in Section 6. A full version of this paper is available, giving details on experiments, more examples, additional methodological remarks, and fully worked out SQL queries [4].

2 Post-mortem Analysis by Queries

Typically, post-mortem checking targets only full (completed) executions stored inside a historical log. We capture a constraint as a query that returns the set of cases incurring in a violation.

Definition 1 (Constraint, Post-mortem Variant). *A* constraint C *is a pair* (Q_{case}, Q_{viol}) *of queries where* Q_{case} *is a* scoping query *that returns all the cases subject to the constraint* C*, while* Q_{viol} *is a* violation detection query *that returns the violating cases such that* Q_{viol} *is always a subset of* Q_{case}*.*

This definition settles our approach for post-mortem checking. It is simply an application of query answering, where the queries are asked against a database instance (representing the execution log) that consists only of completed process instances. In that case, when a tuple $t \in Q_{\text{case}} \setminus Q_{\text{viol}}$, then t represents a case that satisfies the constraint (i.e., $t \in Q_{\text{sat}}$).

Remark 1. Note that an equivalent approach is to represent the constraint as the pair of queries $(Q_{\text{case}}, Q_{\text{sat}})$ instead. The two approaches are interchangeable; sometimes one is simpler to specify, sometimes the other.

Guaranteeing that, for a constraint $(Q_{\text{case}}, Q_{\text{viol}})$, query Q_{viol} always returns a subset of Q_{case} is under the responsibility of the modeler. One way to ensure this is to write Q_{viol} as a query that takes Q_{case} and extends it with a filter to identify violations; however, alternative formulations may be preferred for readability and/or performance needs.

2.1 Database Schema

The structure of the database schema representing the data of the execution log, and how to get a database instance with the data, are important issues. In this paper, we assume these are available. A comprehensive treatment is given by de Murillas et al. [30]. For our purposes of giving illustrating examples, we will simply assume the following two relations in our database, in line with the XES standard extensions [21]:

- `Log`(`CaseId`, `EventId`, `ActivityLabel`, `Timestamp`, `Lifecycle`) which is the main relation.
- An auxiliary `EventData` relation that contains the extra information of the logged events. Events are identified by the combined key (`EventId`, `Lifecycle`), and the extra attributes depend on the application.

2.2 Examples

In the following examples, we assume that the relation `EventData` has the schema (`EventId`, `Lifecycle`, `PackageId`, `CarId`). We also assume that in our processes, we have two activities with the labels “purchase package” and “deliver package”.

Example 1 (Same Shipping Car Constraint). Consider constraint C1 from the Introduction. As we have mentioned before, we have a great flexibility in defining what a violation is (in other words, what is the scope of the constraint). One possibility is to define the cases to be tuples (`CarId`, `Day`, `CountOfDeliveries`). Following this view, the constraint can be represented by the following pair of queries (in favor of saving space, Q_{viol} is only shown, and Q_{case} is merely the same without the last condition in line 6):

```
1 SELECT e.CarId, DATE(l.Timestamp), COUNT(e.PackageId)
2 FROM Log l, EventData e
3 WHERE l.EventId=e.EventId AND l.ActivityLabel='deliver package' AND
4       l.Lifecycle='complete' AND e.Lifecycle='complete'
5 GROUP BY e.CarId, DATE(l.Timestamp)
6 HAVING COUNT(e.PackageId) > 7;
```

A less fine-grained scope, having tuples (`CarId`, `Day`) as our cases, is also possible. In this case, the queries are similar to the ones discussed above, but dropping the third selected column.

Example 1 demonstrates possible queries that define an instance-spanning constraint. To show the uniformity of our approach, the following is an example of an intra-instance constraint.

Example 2 (Shipping Time Constraint). Consider now constraint C2. In what follows, we consider a case to be a package identifier. Again, here we only show Q_{viol}, and Q_{case} is exactly the same without the last condition in line 7.

```
1 SELECT e.PackageId FROM Log l1, Log l2, EventData e
2 WHERE l1.TraceId=l2.TraceId AND l2.EventId=e.EventId AND
3       l1.ActivityLabel='purchase package' AND
4       l2.ActivityLabel='deliver package' AND
5       l1.Lifecycle='complete' AND l2.Lifecycle='complete' AND
6       e.Lifecycle='complete' AND
7       DATE(l2.Timestamp) - DATE(l1.Timestamp) NOT BETWEEN 2 and 5;
```

3 Compliance Monitoring as Incremental View Maintenance

If we want to monitor a constraint *dynamically*, we have to refine our definition. The reason is that the database instance representing the execution log is continuously progressing. Thus, the database instance will contain the data of running (non-completed) process instances along with the completed process instances. Hence, at any moment, any case that is subject to some constraint will be in one of four different states [5,27,11]: 1) a *permanently* violating state; 2) a *permanently* satisfying state; 3) a *currently* violating state that may later be in a satisfying state as a result of the occurrences of new events; and 4) similarly, a *currently* satisfying state that may later be in a violating state. We will refer to the last two states as *pending* states. Note that the set of cases are constantly changing, so new cases can pop up, while others can simply cease to exist. Notice that it depends on the constraint under study whether all such four states

have to be actually considered, or whether instead the constraint only requires a subset thereof. Example 3 discusses a simple constraint such that we can have its cases belonging to the different states.

Regardless of the formal tools, languages, approaches, there is always a "methodology" to go from informal specifications to formal realization.

Example 3 (Monitoring "Followed-By" Constraint). Consider a process that comprises three activities with the labels A, B, and C. Consider the constraint *"Every instance of activity A must be directly followed by an instance of activity B within 20 hours"*. Each process instance is a case here. An instance where A is directly followed by an activity C, say, would be a violation. However, an instance where A is the last event for now, but the instance is not yet completed, is pending violating, as long as A was not longer than 20 hours ago.

In general, we propose:

Definition 2 (Constraint, Compliance monitoring Variant). *A* constraint C *consists of four queries* $(Q_{case}, Q_{viol\text{-}perm}, Q_{viol\text{-}pend}, Q_{sat\text{-}pend})$*, where* Q_{case} *returns all the cases subjected to the constraint* C*, and* $Q_{viol\text{-}perm}$*,* $Q_{viol\text{-}pend}$*, and* $Q_{sat\text{-}pend}$ *return the cases that are permanently violating, pending violating, and pending satisfying, respectively. On any database instance,* $Q_{viol\text{-}perm}$*,* $Q_{viol\text{-}pend}$*, and* $Q_{sat\text{-}pend}$ *always return three mutually exclusive subsets of* Q_{case}*.*

Some of the four queries may be empty, witnessing that the corresponding monitoring state is not compatible with the constraint at hand. Also, the permanently satisfying cases can be derived from the four queries as $Q_{\text{sat-perm}} = Q_{\text{case}} - (Q_{\text{viol-perm}} \cup Q_{\text{viol-pend}} \cup Q_{\text{sat-pend}})$. Typically, the query Q_{viol} in the post-mortem checking variant corresponds to the union of $Q_{\text{viol-perm}}$ and $Q_{\text{viol-pend}}$ in the compliance monitoring variant. Similarly, the query Q_{sat} corresponds to the pair $Q_{\text{sat-perm}}$ and $Q_{\text{sat-pend}}$.

Example 4 (Monitoring Same Shipping Car Constraint). Consider again constraint C1. Queries Q_{case} and $Q_{\text{viol-perm}}$ are the same as Q_{case} and Q_{viol} of Example 1. Then for $Q_{\text{sat-pend}}$, we use the following query:

```
SELECT e.CarId, DATE(l.Timestamp), COUNT(e.PackageId)
FROM Log l, EventData e
WHERE l.EventId=e.EventId AND l.ActivityLabel='deliver package' AND
      l.Lifecycle='complete' AND e.Lifecycle='complete' AND
      DATE(l.Timestamp)=CURRENT_DATE
GROUP BY e.CarId, DATE(l.Timestamp)
HAVING COUNT(e.PackageId) <= 7;
```

The fourth query, $Q_{\text{viol-pend}}$, will always be empty for this constraint, because once a shipping car has been used too often, it is immediately a permanent violation, since the situation cannot be salvaged anymore.

4 Experiments

Once constraints are expressed as queries per our methodology, compliance monitoring becomes an application of view maintenance. DBToaster is a state-of-the-art incremental query processor [23,24,34]. As a proof-of-concept of our approach,

we tested DBToaster on constraints from the work of Winter et al. on automatic discovery of ISCs [36]. We have also used the execution logs provided by these authors as sample input data [10]. To manage our experiments, we performed some preprocessing steps (detailed in the full version [4]).

The tested constraints are expressed over the three processes whose models are shown in Figure 1. Any order has a corresponding initiated "Bill" process. Moreover, printers are considered a shared resource between all the processes.

We have specifically run tests on five constraints ISC1, ISC2a, ISC2b, ISC3, and ISC4 from Winter et al. Due to space limitations, here, we only present results on the following three constraints:

ISC1 There is exactly one delivery activity per day in which all the finished orders/bills of that day so far are delivered to the post office simultaneously.

ISC2a All print jobs must be completed within 10 minutes in at least 95% of all cases per month.

ISC3 If a flyer or poster order is received $P2$ is started afterwards. Moreover, the corresponding bill process must be started before the order is delivered to the post office.

We slightly modified the original constraints [36] to better match with the log data [10]. Each constraint was expressed using SQL queries, according to Definition 2 [4].

Running Time. We measured the running time of the five monitored constraints, averaged over 10 runs. The time is measured after every 300 insertions for a total of 30636 insertions (the number of events in the dataset). This experiment was performed on a personal laptop running macOS 12.2.1 with RAM of 16 GB and processor speed of 2.6Hz.

The attentive reader may note that the line sometimes goes down. This is an artifact of the way in which we avoided a bug in the Scala version of DBToaster. We reran DBToaster on ever larger sequences of insertions; the queries the system uses internally to retrieve the current snapshot are sometimes slightly faster on slightly larger database instances. The outcome of this experiment produces the

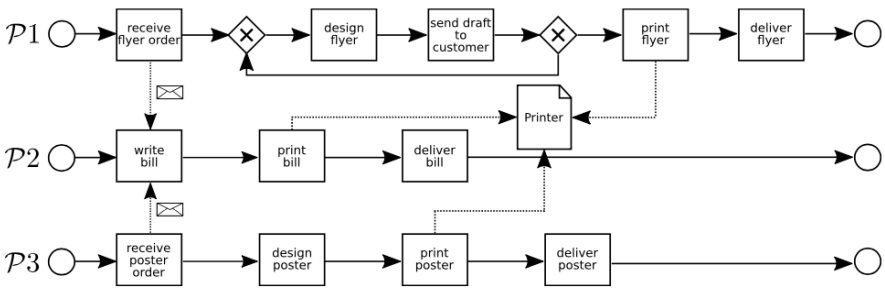

Fig. 1: Processes: Flyer Order, Bill, and Poster Order [36].

average time needed, per event, to maintain the queries defining the constraint. We can see that the slope is significantly higher for the first constraint; indeed, this constraint requires rather complex SQL queries. For ISC1, the maintenance time is less than half a millisecond; for ISC3, less than 1/6th of a millisecond; and for the other constraints less than 1% of a millisecond.

Queries Result Sizes. Figure 3 shows query result sizes (the number of cases) relative to time (number of insertions). This shows how cases are changing their status (pending or permanent, violating or satisfying). The query size is reported every 500 insertions in the case of ISC1, every 100 insertions for ISC2a, as it displays a more fine-grained behavior.

Tracing Cases. As an illustration of the feasibility of our approach and its compatibility with monitoring on a very detailed level, we show in Figure 4 the evolution in status of some individual cases of ISC1 over time.

5 Sequence Data Extensions of Query Languages

We have mentioned before that any data model with a sufficiently expressive query language can be used to express the constraints. Although, we chose to work with the relational data model with SQL for the reasons we mentioned, it is interesting to briefly discuss query languages for the relational data model extended with sequences [25,33]. Indeed, a trace is a sequence of events. Hence, representing the relative order of the events is quite natural in a sequence data model. This level of abstraction, of viewing traces as sequences of abstract events, is often assumed when working with temporal and dynamic logics [16,31,17].

Sequence Datalog [3,6,28] is an extension of the query language Datalog, to work with sequences as first class citizens. We will briefly showcase this language by considering an example of a constraint that is usually handled with temporal logic.

Example 5 (Strict Sequencing [16]). Let `a` and `b` be two activities. There is a *strict sequencing* relation [2] between `a` and `b` if the log satisfies the following:
- there exists a trace where `a` is immediately followed by `b`; and
- there are not any traces where `b` is immediately followed by `a`.

There are two possible violations of this constraint: (1) not having a trace with `b` directly following `a`; (2) having a trace with `a` directly following `b`.

For the purpose of expressing this constraint, assume we have the relation `Log(TraceId, Events)`, where `Events` are sequences of labels of activities. Then, this constraint can be expressed by the following Sequence Datalog program.

```
a_before_b()← Log(@traceId, $pre.a.b.$post).
violation() ← ¬a_before_b(). % violation (1)
violation() ← Log(@traceId, $pre.b.a.$post). % violation (2)
```

This program illustrates a number of Sequence Datalog features:
- the dot is the concatenation operator.

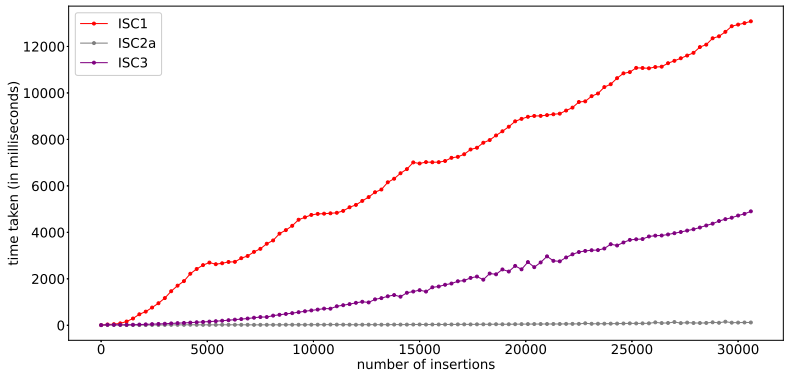

Fig. 2: Running time taken to monitor the constraints.

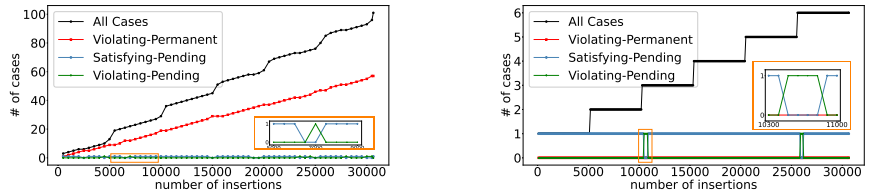

Fig. 3: Result sizes of queries for ISC1 and ISC2a. Since our measurements consist of 600 data points (even 3000 for ISC2a), the plots are at rather large scale. To show more detail, we provide insets that zoom in on selected regions (orange rectangles).

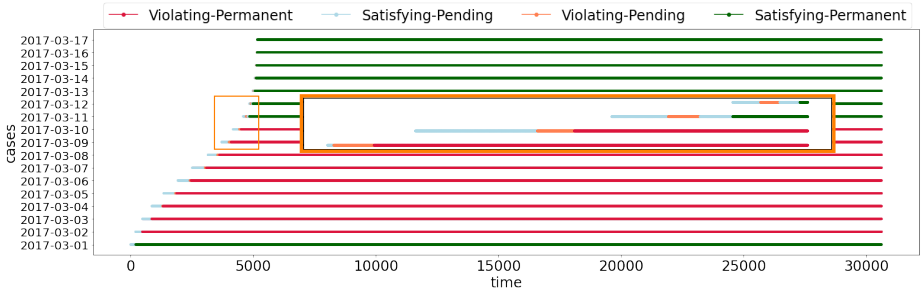

Fig. 4: Some ISC1 cases (days of March) and how each is changing its status through time. Here the measurement consists of 30600 data points per case, so the plot is at a very large scale. The inset shows more detail by zooming on the selected region (orange rectangle).

- `@traceId` is an *atomic* variable (indicated by the `@` symbol) representing atomic values (in this case, trace identifiers).
- `$pre` and `$post` are *sequence* variables (indicated by the `$` symbol) representing (possibly empty) sequences of atomic values.

The utility of using Sequence Datalog can be appreciated if we compare the above program with the same query expressed in SQL which is more complicated and much longer [4]. This paves the way towards the adoption of Sequence Datalog to express, check and monitor constraints, once techniques for (incremental) query processing will be implemented.

6 Discussion

In this paper, we have looked into the problems of post-mortem checking and compliance monitoring of constraints over business processes. Specifically, we focused on ISCs as recently introduced in the process mining field, and caught attention since it refers to complex constraints that span multiple process instances. Although there have been extensive works on inventorying and categorizing ISCs [32,36], a crisp definition of what is or is not an ISC, however, seems to be elusive. Indeed, the notion of constraint is so broad that we propose here to *define* any constraint as two or four queries posed against the database instance that represents a (partial) execution log. This approach gives us huge flexibility, and moreover, allows the use of incremental query processing techniques out of the box.

In using the DBToaster system for our experiments, we faced a few technical issues. The main challenge was that the Scala version of DBToaster gets stuck when retrieving snapshots over the course of the insertions. Another limitation is that SQL is not yet fully supported, although complex queries can be expressed. This required us to sometimes rewrite queries in equivalent form. Finally, some built-in functions (e.g., on strings or dates) are missing from the Scala version. Thus, our experiments should be seen more as a proof-of-concept of the feasibility of our approach.

In this discussion, we briefly touch upon the main difference between our approach and the main approach to monitoring ISCs, based on the Event Calculus (EC) [26,29,22]. Most monitoring systems based on EC are implemented using Prolog. Using EC to express a constraint seems to be very *procedural* albeit being defined in logical programming language. For example, to monitor a constraint such as C1, in EC one would define a rule that increments a counter every time a delivery event occurs. At the end, that counter value should be at most seven as per the constraint. A similar approach was followed in the paper by Montali et al. [29] to monitor intra-instance constraints with EC. Events come in time, and Prolog rules that fire every new time instant, are used to check various constraints dynamically. However, these incremental rules are manually implemented. On the contrary, using an incremental query processor shifts the focus on what the queries (or constraints) themselves are rather than what the rules

are that are responsible for this incremental maintenance. Hence, our approach is more declarative.

At the end of this discussion, we mention a few points for further research. Since there are some algorithms that are used to discover ISCs from execution logs [36], and these algorithms search for explicit patterns, one could define a common language to report the results of those algorithms and use those results to automatically write the SQL queries monitoring each of the reported constraints. Thus, the whole process could be automated. An engineering issue is to investigate which discipline of query formulation works best with incremental view maintenance methods. Another natural continuation of this work is to explore the possibility of monitoring object-centric processes where multiple interrelated objects are co-evolved. Our approach can be readily applied to this setting, considering that our techniques operate over a full-fledged relational database.

Acknowledgments. We thank Stefanie Rinderle-Ma and Jürgen Mangler for initial discussions. Heba Aamer is supported by the Special Research Fund (BOF) (BOF19OWB16). Marco Montali is partially supported by the PRIN 2022 PINPOINT project.

References

1. van der Aalst, W.M.P.: Process Mining: Overview and Opportunities. ACM Trans. Manag. Inf. Syst. **3**(2), 7:1–7:17 (2012). https://doi.org/10.1145/2229156.2229157
2. van der Aalst, W.M.P., Weijters, T., et al.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004).
3. Aamer, H., Hidders, J., Paredaens, J., Van den Bussche, J.: Expressiveness within Sequence Datalog. PODS 2021.
4. Aamer, H., Montali, M., Van den Bussche, J.: What Can Database Query Processing Do for Instance-Spanning Constraints? arXiv:2206.00140 (2022)
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
6. Bonner, A., Mecca, G.: Sequences, Datalog, and Transducers. J. Comput. Syst. Sci. **57**, 234–259 (1998)
7. Calvanese, D., De Giacomo, G., Montali, M., Patrizi, F.: Verification and monitoring for first-order LTL with persistence-preserving quantification over finite and infinite traces. IJCAI-ECAI 2022.
8. Cardoso, E., Montali, M., Calvanese, D.: Representing and querying norm states using temporal ontology-based data access. EDOC 2019.
9. Chirkova, R., Yang, J.: Materialized Views. Foundations and Trends in Databases **4**(4), 295–405 (2012). https://doi.org/10.1561/1900000020
10. CRISP project at Universität Wien: Logs Webpage. `http://gruppe.wst.univie.ac.at/projects/crisp/index.php?t=discovery`, accessed: 2022-04-29
11. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. BPM 2014.
12. De Giacomo, G., De Masellis, R., Maggi, F.M., Montali, M.: Monitoring constraints and metaconstraints with temporal logics on finite traces. TOCEM 2022.
13. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. on Computational Logic **10**(3) (2009)

14. Esser, S., Fahland, D.: Multi-Dimensional Event Data in Graph Databases. J. Data Semant. **10**(1), 109–141 (2021). https://doi.org/10.1007/s13740-021-00122-1
15. Fdhila, W., Gall, M., Rinderle-Ma, S., Mangler, J., Indiono, C.: Classification and Formalization of Instance-Spanning Constraints in Process-Driven Applications. BPM 2016.
16. Giacomo, G.D., Felli, P., Montali, M., Perelli, G.: HyperLDLf: a Logic for Checking Properties of Finite Traces Process Logs. IJCAI 2021.
17. Giacomo, G.D., Masellis, R.D., et al.: Monitoring Business Metaconstraints Based on LTL and LDL for Finite Traces. BPM 2014.
18. Gupta, A., Mumick, I.S. (eds.): Materialized Views: Techniques, Implementations, and Applications. MIT Press, Cambridge, MA, USA (1999)
19. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Eng. Bull. **18**(2), 3–18 (1995).
20. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining Views Incrementally. SIGMOD 1993.
21. IEEE 1849-2016 XES Standard. `https://www.xes-standard.org/`.
22. Indiono, C., Mangler, J., et al.: Rule-Based Runtime Monitoring of Instance-Spanning Constraints in Process-Aware Information Systems. OTM 2016.
23. Kennedy, O., et al.: DBToaster: Agile Views for a Dynamic Data Management System. CIDR 2011.
24. Koch, C., et al.: DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. VLDB J. **23**(2), 253–278 (2014).
25. LDBC Graph Query Language Task Force: G-CORE: A core for future graph query languages. SIGMOD 2018.
26. Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., van der Aalst, W.M.P.: Compliance Monitoring in Business Processes: Functionalities, Application, and Tool-Support. Inf. Syst. **54**, 209–234 (2015). https://doi.org/10.1016/j.is.2015.02.007
27. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. BPM 2011.
28. Mecca, G., Bonner, A.: Query Languages for Sequence Databases: Termination and Complexity. IEEE TKDE **13**(3), 519–525 (2001)
29. Montali, M., et al.: Monitoring Business Constraints with the Event Calculus. ACM Trans. Intell. Syst. Technol. **5**(1), 17:1–17:30 (2013).
30. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. Softw. Syst. Model. **18**(2), 2019.
31. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. EDOC 2007.
32. Rinderle-Ma, S., Gall, M., Fdhila, W., Mangler, J., Indiono, C.: Collecting Examples for Instance-Spanning Constraints. arXiv:1603.01523 (2018)
33. Shen, W., et al.: Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. VLDB 2007.
34. DBToaster Webpage. `https://dbtoaster.github.io/index.html`.
35. Warner, J., Atluri, V.: Inter-instance Authorization Constraints for Secure Workflow Management. SACMAT 2006.
36. Winter, K., et al.: Discovering Instance and Process Spanning Constraints from Process Execution Logs. Inf. Syst. **89**, 101484 (2020).