

Data-Aware Process Oriented Query Language



Eduardo Gonzalez Lopez de Murillas, Hajo A. Reijers,
and Wil M. P. van der Aalst

Abstract The size of execution data available for process mining analysis grows several orders of magnitude every couple of years. Extracting and selecting the relevant data to enable process mining remains a challenging and time-consuming task. In fact, it is the biggest handicap when applying process mining and other forms of process-centric analysis. This work presents a new query language, DAPOQ-Lang, which overcomes some of the limitations identified in the field of process querying and fits within the Process Querying Framework. The language is based on the OpenSLEX meta model, which combines both data and process perspectives. It provides simple constructs to intuitively formulate questions. The syntax and semantics have been formalized and an implementation of the language is provided, along with examples of queries to be applied to different aspects of the process analysis.

1 Introduction

One of the main goals of process mining techniques is to obtain insights into the behavior of systems, companies, business processes, or any kind of workflow under study. Obviously, it is important to perform the analysis on the right data. Data extraction and preparation are among the first steps to take and, in many cases, up to 80% of the time and effort, and 50% of the cost is spent during these phases [12].

E. G. L. de Murillas (✉)

Department of Mathematics and Computer Science, Eindhoven University of Technology,
Eindhoven, The Netherlands

H. A. Reijers

Department of Information and Computing Sciences, Utrecht University, Utrecht, The
Netherlands

e-mail: h.a.reijers@uu.nl

W. M. P. van der Aalst

Department of Computer Science, RWTH Aachen University, Aachen, Germany

e-mail: wvdaalst@pads.rwth-aachen.de

© Springer Nature Switzerland AG 2022

A. Polyvyanyy (ed.), *Process Querying Methods*,

https://doi.org/10.1007/978-3-030-92875-9_3

Being able to extract and query some specific subset of the data becomes crucial when dealing with complex and heterogeneous datasets. In addition, the use of querying tools allows one to find specific cases or exceptional behavior. Whatever the goal, analysts often find themselves in the situation in which they need to develop ad hoc software to deal with specific datasets, given that existing tools might be difficult to use, too general, or just not suitable for process analysis.

Different approaches exist to support the *querying of process data*. Some of them belong to the field of business process management (BPM). In this field, events are the main source of information. They represent transactions or activities that were executed at a certain moment in time in the environment under study. Querying this kind of data allows us to obtain valuable information about the behavior and execution of processes. There are other approaches originating from the field of data provenance, which are mainly concerned with recording and observing the origins of data. This field is closely related to scientific workflows in which the traceability of the origin of experimental results becomes crucial to guarantee correctness and reproducibility. In the literature, we find many languages to query process data. However, none of these approaches succeeds at combining process and data aspects in an integrated way. An additional challenge to overcome is the development of a query mechanism that allows to exploit this combination, while being intuitive and easy to use.

In order to make the querying of process event data easier and more efficient, we propose the Data-Aware Process Oriented Query Language (DAPOQ-Lang). This query language, first introduced in [3], exploits both process and data perspectives. The aim of DAPOQ-Lang is not to theoretically enable new types of computations, but to ease the task of writing queries in the specific domain of process mining. Therefore, our focus is on the ease of use. We propose the following example to show the ease of use of DAPOQ-Lang. Let us consider a generic question that could be asked by an analyst when carrying out a process mining project:

GQ: In which cases, there was (a) an event that happened between time T1 and T2, (b) that performed a modification in a version of class C, (c) in which the value of field F changed from X to Y?

This query involves several types of elements: cases, events, object versions, and attributes. We instantiate this query with some specific values for T1 = “1986/09/17 00:00”, T2 = “2016/11/30 19:44”, C = “CUSTOMER”, F = “ADDRESS”, X = “Fifth Avenue”, and Y = “Sunset Boulevard”. Query 1 presents the corresponding DAPOQ-Lang query. This example shows how compact a DAPOQ-Lang query can be. The specifics of this query will be explained in the coming sections.

The rest of this chapter is organized as follows. Section 2 introduces some background information, which is needed to understand the specifics of our query language. Section 3 presents the query language, focusing on the syntax and semantics. Section 4 provides information about the implementation and its evaluation. Section 5 presents possible use cases. Section 6 positions DAPOQ-Lang in the Process Querying Framework [10]. Section 7 concludes the chapter.

Query 1 DAPOQ-Lang query to retrieve cases with an event happening between two dates that changed the address of a customer from “Fifth Avenue” to “Sunset Boulevard”.

```

1  def P1 = createPeriod("1986/09/17_00:00", "2016/11/30_19:44", "yyyy/MM/dd_HH:mm
    ↪ ")
2
3  casesOf(
4    eventsOf(
5      versionsOf(
6        allClasses().where {name == "CUSTOMER"}
7      ).where {
8        changed([at: "ADDRESS", from:"Fifth_Avenue", to:"Sunset_Boulevard"]) }
9    ).where
10   {
11     def P2 = createPeriod(it.timestamp)
12     during(P2, P1)
13   }
14 )

```

2 Preliminaries

To enable our approach to data querying, we need to have access to data storage, and the information should comply with a certain structure. An appropriate structure has been previously defined as a meta model [4] and implemented in a *queryable* file format called OpenSLEX. The meta model captures all the necessary aspects to enable data querying with our language. This section describes the structure of OpenSLEX and provides the necessary background to understand the details of DAPOQ-Lang.

Standards of reference, like XES [5], are focused on the process view (events, traces, and logs) of systems. OpenSLEX supports all concepts present in XES but, in addition, considers the data elements (data model, objects, and versions) as an integral part of its structure. This makes it more suitable for database environments where only a small part of the information is process-oriented (i.e., events) with respect to the rest of data objects of different classes that may be seen as an augmented view on the process information. The OpenSLEX format is supported by a meta model (Fig. 1) that considers *data models* and *processes* as the entities at the highest abstraction level. These entities define the structure of more granular elements like *logs*, *cases*, and *activity instances* with respect to processes and *objects* with respect to classes in the data model. Each of these elements at the intermediate level of abstraction can be broken apart into more granular pieces. This way, *cases* are formed by *events*, and *objects* can be related to several *object versions*. Both *events* and *object versions* represent different states of a higher level abstraction (*cases* or *objects*) at different points in time. A detailed ER diagram of the OpenSLEX format can be found online.¹ The format makes use of an SQL

¹ <https://github.com/edugonza/OpenSLEX/blob/master/doc/meta-model.png>.

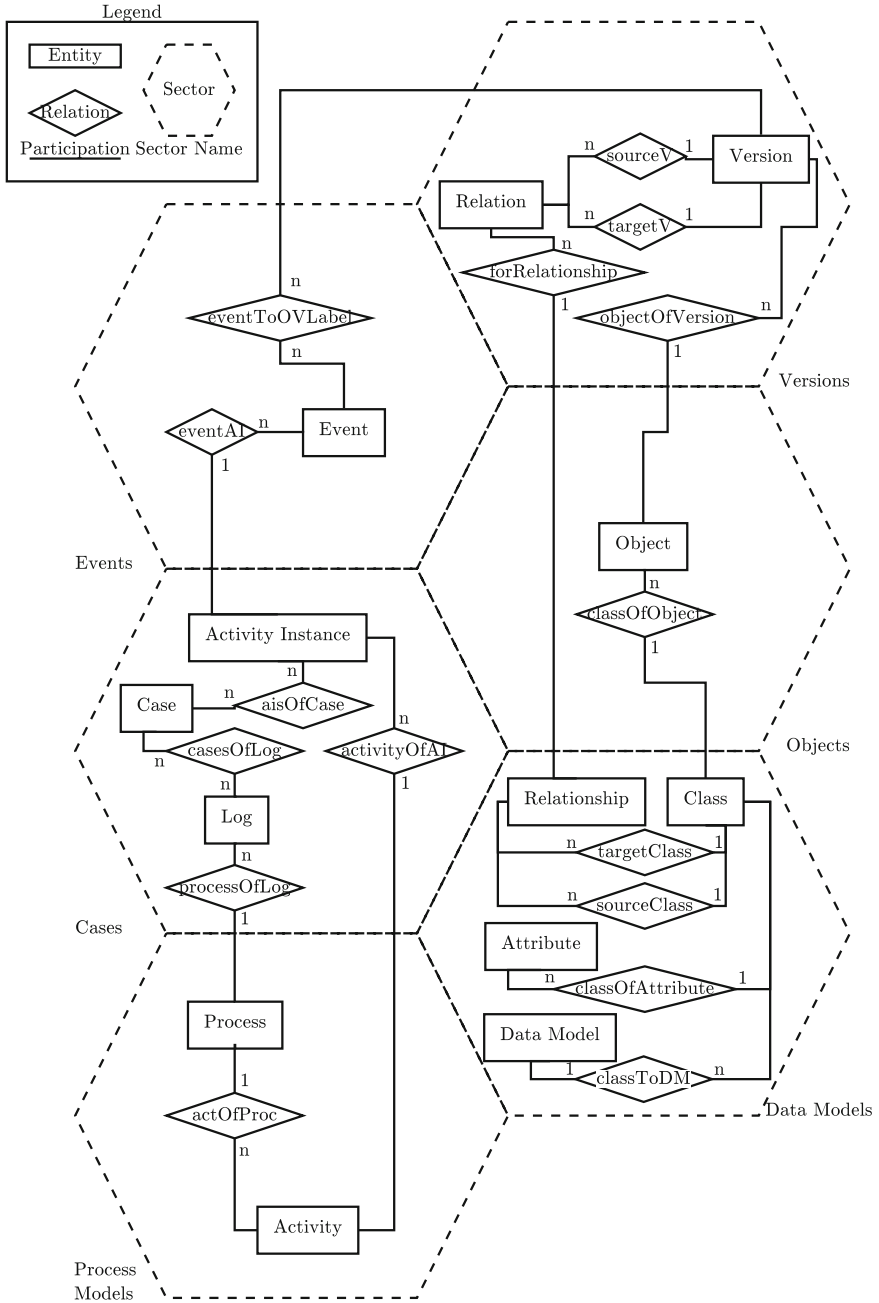


Fig. 1 ER diagram of the OpenSLEX meta model. The entities have been grouped into sectors, delimited by the dashed lines

schema to store all the information, and a Java API² is available for its integration in other tools. The use of OpenSLEX in several environments, e.g., database redo logs and ERP databases, is evaluated in [4], focusing on the data extraction and transformation phase. To provide the necessary background for the understanding of this work, a simplified version of the meta model is formally presented below. Every database system contains information structured with respect to a data model. Definition 1 provides a formalization of a data model in the current context.

Definition 1 (Data Model) A data model is a tuple $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$, such that

- CL is a set of class names.
- AT is a set of attribute names.
- $classOfAttribute \in AT \rightarrow CL$ is a function that maps each attribute to a class.
- RS is a set of relationship names.
- $sourceClass \in RS \rightarrow CL$ is a function mapping each relationship to its source class.
- $targetClass \in RS \rightarrow CL$ is a function mapping each relationship to its target class.

Data models contain classes (i.e., tables), which contain attribute names (i.e., columns). Classes are related by means of relationships (i.e., foreign keys). Definition 2 formalizes each of the entities of the OpenSLEX meta model, as can be observed in Fig. 1, and shows connections between them.

Definition 2 (Connected Meta Model) Let V be some universe of values and TS a universe of timestamps. A connected meta model is defined as a tuple $CMM = (DM, OC, classOfObject, OVC, objectOfVersion, EC, eventToOVLabel, IC, eventAI, PMC, activityOfAI, processOfLog)$ such that

- $DM = (CL, AT, classOfAttribute, RS, sourceClass, targetClass)$ is a data model.
- OC is a collection of objects.
- $classOfObject \in OC \rightarrow CL$ is a function that maps each object to its corresponding class.
- $OVC = (OV, attValue, startTimestamp, endTimestamp, REL)$ is a version collection, where OV is a set of object versions, $attValue \in (AT \times OV) \dashrightarrow V$ is a mapping of pairs of object version and attribute to a value, $startTimestamp \in OV \rightarrow TS$ is a mapping between object versions and start timestamps, $endTimestamp \in OV \rightarrow TS$ is a mapping between object versions and end timestamps, and $REL \subseteq (RS \times OV \times OV)$ is a set of triples relating pairs of object versions through a specific relationship.
- $objectOfVersion \in OV \rightarrow OC$ is a function that maps each object version to an object.
- $EC = (EV, EVAT, eventTimestamp, eventLifecycle, eventResource, eventAttributeValue)$ is an event collection, where EV is a set of events, $EVAT$

² <https://github.com/edugonza/OpenSLEX>.

is a set of event attribute names, $eventTimestamp \in EV \rightarrow TS$ maps events to timestamps, $eventLifecycle \in EV \rightarrow \{start, complete, \dots\}$ maps events to life cycle attributes, $eventResource \in EV \rightarrow V$ maps events to resource attributes, and $eventAttributeValue \in (EV \times EVAT) \not\rightarrow V$ maps pairs of event and attribute name to values.

- $eventToOVLabel \in (EV \times OV) \not\rightarrow V$ is a function that maps pairs of an event and an object version to a label. The existence of a label associated with an event and an object version, i.e., $(ev, ov) \in dom(eventToOVLabel)$, means that both event and object version are linked. The label defines the nature of the link, e.g., “insert”, “update”, “delete”, etc.
- $IC = (AI, CS, LG, aisOfCase, casesOfLog, CSAT, caseAttributeValue, LGAT, logAttributeValue)$ is an instance collection, where AI is a set of activity instances, CS is a set of cases, LG is a set of logs, $aisOfCase \in CS \rightarrow \mathcal{P}(AI)$ is a mapping between cases and sets of activity instances,³ $casesOfLog \in LG \rightarrow \mathcal{P}(CS)$ is a mapping between logs and sets of cases, $CSAT$ is a set of case attribute names, $caseAttributeValue \in (CS \times CSAT) \not\rightarrow V$ maps pairs of case and attribute name to values, $LGAT$ is a set of log attribute names, and $logAttributeValue \in (LG \times LGAT) \not\rightarrow V$ maps pairs of log and attribute name to values.
- $eventAI \in EV \rightarrow AI$ is a function that maps each event to an activity instance.
- $PMC = (PM, AC, actOfProc)$ is a process model collection, where PM is a set of processes, AC is a set of activities, and $actOfProc \in PM \rightarrow \mathcal{P}(AC)$ is a mapping between processes and sets of activities.
- $activityOfAI \in AI \rightarrow AC$ is a function that maps each activity instance to an activity.
- $processOfLog \in LG \rightarrow PM$ is a function that maps each log to a process.

A connected meta model (CMM) provides the functions that make it possible to connect all the entities in the meta model. This is important in order to correlate elements, e.g., events that modified the same object. However, some constraints must be fulfilled for a meta model to be considered a valid connected meta model (e.g., versions of the same object do not overlap in time). The details about such constraints are out of the scope of this work, but their description can be found in [4]. DAPOQ-Lang queries are executed on data structures that fulfill the constraints set on the definition of a connected meta model. According to our meta model description, *events* can be linked to *object versions*, which are related to each other by means of *relations*. These *relations* are instances of data model *relationships*. In database environments, this would be the equivalent to using foreign keys to relate table rows and knowing which events relate to each row. For the purpose of this work, we assume that pairwise correlations between events, by means of related object versions, are readily available in the input connected meta model.

³ $\mathcal{P}(X)$ is the powerset of X , i.e., $Y \in \mathcal{P}(X)$ if $Y \subseteq X$.

3 DAPOQ-Lang

DAPOQ-Lang is a Data-Aware Process Oriented Query Language that allows the user to query data and process information stored in a structure compatible with the OpenSLEX meta model [4]. As described in Sect. 2, OpenSLEX combines database elements (data models, objects, and object versions) with common process mining data (events, logs, and processes), considering them as first-class citizens. DAPOQ-Lang considers the same first-class citizens as OpenSLEX, which makes it possible to write queries in the process mining domain enriched with data aspects with lower complexity than in other general purpose query languages like SQL.

Intuitively, we could think that considering all the elements of the OpenSLEX meta model as first-class citizens would introduce a lot of complexity in our language. However, these elements have been organized in a type hierarchy as subtypes of higher level superclasses (Fig. 2). It can be seen that *MMElement* (Meta Model Element) is an abstract class at the highest level *superclass*. Next, we distinguish two subtypes of elements: (1) stored elements (*StoredElement*), i.e., elements that can be found directly stored in an OpenSLEX structure, such as activities, events, objects, and logs and (2) computed elements (*ComputedElements*), i.e., elements that are computed based on the rest, temporal periods of cases and temporal periods of events. We will exploit this hierarchy to design a simple language, providing many basic functions that can operate on any *MMElement*, and some specific functions that focus on specific subtypes. Given a connected meta model *CMM* (Definition 2), we define the concept of *MMElement* in DAPOQ-Lang as the union of all its terminal subtypes: $MMElement = AC \cup LG \cup EV \cup REL \cup OC \cup AT \cup CL \cup PER \cup PM \cup CS \cup AI \cup OV \cup RS \cup DM$. Some of the functions that we define operate on sets of elements ($\mathcal{P}(MMElement)$). However, as a constraint of our query language, we require that the elements of those sets belong to the same subtype (i.e., a set of *Class* elements, or a set of *Version* elements, or a set of *Event* elements, etc.). Therefore, we define the set *MMSets* as the set of all possible subsets of each element type in a

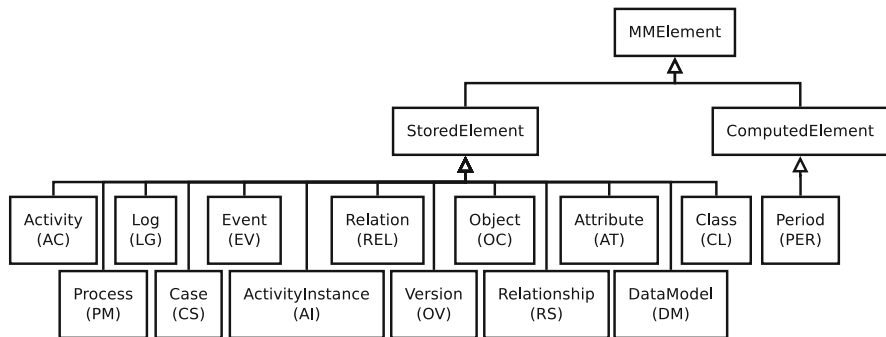


Fig. 2 DAPOQ-Lang type hierarchy. Arrows indicate subtype relations

meta model MM :

$$MMSets = \mathcal{P}(AC) \cup \mathcal{P}(LG) \cup \mathcal{P}(EV) \cup \mathcal{P}(REL) \cup \mathcal{P}(OC) \cup \mathcal{P}(AT) \cup \mathcal{P}(CL) \cup \mathcal{P}(PER) \cup \mathcal{P}(PM) \cup \mathcal{P}(CS) \cup \mathcal{P}(AI) \cup \mathcal{P}(OV) \cup \mathcal{P}(RS) \cup \mathcal{P}(DM) \quad (1)$$

The output of any query will be an element set $es \in MMSets$, i.e., a set of elements of the same type. The following subsections describe the syntax and semantics of DAPOQ-Lang in detail.

3.1 Syntax

The language has been designed with the aim to find a balance between simplicity and expressive power. To do so, we exploited the specifics of the underlying meta model defining a total of 57 basic functions, as organized in 5 well-defined blocks, that can be applied in the context of a given meta model MM . The functions proposed in Sects. 3.1.1–3.1.5 will be used to define syntax and semantics of DAPOQ-Lang in Sects. 3.1 and 3.2.

3.1.1 Terminal Meta Model Elements

We define a set of 13 basic terminal functions. Each of them maps to the set of all elements of the corresponding type (Fig. 2) in the OpenSLEX meta model structure (Definition 2). Given a connected meta model, we define the following:

1. *allDatamodels*: the set of all data models, i.e., DM .
2. *allClasses*: the set of all classes, i.e., CL .
3. *allAttributes*: the set of all class attributes, i.e., AT .
4. *allRelationships*: the set of all class relationships, i.e., RS .
5. *allObjects*: the set of all objects, i.e., OC .
6. *allVersions*: the set of all object versions, i.e., OV .
7. *allRelations*: the set of all relations, i.e., REL .
8. *allEvents*: the set of all events, i.e., EV .
9. *allActivityInstances*: the set of all activity instances, i.e., AI .
10. *allCases*: the set of all cases, i.e., CS .
11. *allLogs*: the set of all logs, i.e., LG .
12. *allActivities*: the set of all activities, i.e., AC .
13. *allProcesses*: the set of all processes, i.e., PM .

3.1.2 Elements Related to Elements



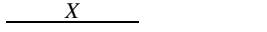

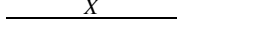
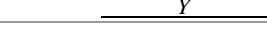
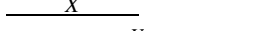
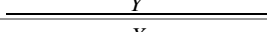
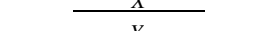
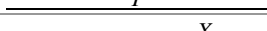
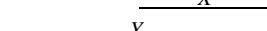
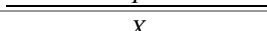
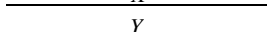
The following 14 functions take as an input a set of elements of the same type and return a set of elements related to them of the type corresponding to the return type of the function, e.g., a call to *eventsOf(es)*, being $es \in \mathcal{P}(LG)$ will return the set of events that are related to the logs in the set es . Thanks to the subtype hierarchy, in most of the cases, we can reuse the same function call for input sets of any type, which leads to a more compact syntax. In the cases when an input of any type would not make sense, we can still restrict the input type to a particular kind, as is the case with the function *versionsRelatedTo*, which only accepts sets of object versions as input.

14. *datamodelsOf* $\in MMSets \rightarrow \mathcal{P}(DM)$: returns the set of data models related to the input.
15. *classesOf* $\in MMSets \rightarrow \mathcal{P}(CL)$: returns the set of classes related to the input.
16. *attributesOf* $\in MMSets \rightarrow \mathcal{P}(A)$: returns the set of attributes related to the input.
17. *relationshipsOf* $\in MMSets \rightarrow \mathcal{P}(RS)$: returns the set of relationships related to the input.
18. *objectsOf* $\in MMSets \rightarrow \mathcal{P}(O)$: returns the set of objects related to the input.
19. *versionsOf* $\in MMSets \rightarrow \mathcal{P}(OV)$: returns the set of object versions related to the input.
20. *relationsOf* $\in MMSets \rightarrow \mathcal{P}(REL)$: returns the set of relations related to the input.
21. *eventsOf* $\in MMSets \rightarrow \mathcal{P}(E)$: returns the set of events related to the input.
22. *activityInstancesOf* $\in MMSets \rightarrow \mathcal{P}(AI)$: returns the set of activity instances related to the input.
23. *activitiesOf* $\in MMSets \rightarrow \mathcal{P}(AC)$: returns the set of activities related to the input.
24. *casesOf* $\in MMSets \rightarrow \mathcal{P}(CS)$: returns the set of cases related to the input.
25. *logsOf* $\in MMSets \rightarrow \mathcal{P}(LG)$: returns the set of logs related to the input.
26. *processesOf* $\in MMSets \rightarrow \mathcal{P}(PM)$: returns the set of processes related to the input.
27. *versionsRelatedTo* $\in \mathcal{P}(OV) \rightarrow \mathcal{P}(OV)$: returns the set of object versions directly related (distance 1) to the input object versions through any kind of relationship.

3.1.3 Computation of Temporal Values

Some elements in our meta model contain temporal properties (e.g., events have timestamps, object versions have life spans, etc.) which allows making temporal computations with them. To do so, we provide the following 8 functions to compute time periods (with a start and an end), as well as durations. Durations (*DUR*) are special in the sense that they can be considered as a scalar and are not part of the

Table 1 Relations in Allen's interval algebra

Relation	Name	Illustration	Interpretation
$X < Y$	before		X takes place before Y
$Y > X$	after		
$X \text{ m } Y$	meets		X meets Y (<i>i</i> stands for <i>inverse</i>)
$Y \text{ mi } X$	meetsInv		
$X \text{ o } Y$	overlaps		X overlaps with Y
$Y \text{ oi } X$	overlapsInv		
$X \text{ s } Y$	starts		X starts Y
$Y \text{ si } X$	startsInv		
$X \text{ d } Y$	during		X during Y
$Y \text{ di } X$	duringInv		
$X \text{ f } Y$	finishes		X finishes Y
$Y \text{ fi } X$	finishesInv		
$X = Y$	matches		X is equal to Y

MMElement subtype hierarchy. Durations can only be used to be compared with other durations.

28. $periodsOf \in MMSets \rightarrow \mathcal{P}(PER)$: returns the computed periods for each of the elements of the input set.
29. $globalPeriodOf \in MMSets \rightarrow PER$: returns a global period for all the elements in the input set, i.e., the period from the earliest to the latest timestamp.
30. $createPeriod \in TS \times TS \rightarrow PER$: returns a period for the specified start and end timestamps.
31. $getDuration \in PER \rightarrow DUR$: returns the duration of a period in milliseconds.
32. $Duration.ofSeconds \in \mathbb{N} \rightarrow DUR$: returns the duration of the specified seconds.⁴
33. $Dduration.ofMinutes \in \mathbb{N} \rightarrow DUR$: returns the duration of the specified minutes.
34. $Duration.ofHours \in \mathbb{N} \rightarrow DUR$: returns the duration of the specified hours.
35. $Duration.ofDays \in \mathbb{N} \rightarrow DUR$: returns the duration of the specified days.

3.1.4 Temporal Interval Algebra

Allen's interval algebra, described in [1], introduces a calculus for temporal reasoning that defines possible relations between time intervals. It provides the tools to reason about the temporal descriptions of events in the broadest sense. Table 1 shows the 13 base relations between two intervals. These temporal relations

⁴ \mathbb{N} is the set of natural numbers.

are used in our approach to reason about data elements for which we can compute a temporal interval.

We have introduced the functions to compute and create periods of time. The following 13 functions cover all the interval operators, described by Allen's interval algebra, that we can use to compare periods. Take (a, b) to be a pair of periods for which:

36. $before \in PER \times PER \rightarrow \mathbb{B}$: $before(a, b)$ iff a takes place before b .⁵
37. $after \in PER \times PER \rightarrow \mathbb{B}$: $after(a, b)$ iff a takes place after b .
38. $meets \in PER \times PER \rightarrow \mathbb{B}$: $meets(a, b)$ iff the end of a is equal to the start of b .
39. $meetsInv \in PER \times PER \rightarrow \mathbb{B}$: $meetsInv(a, b)$ iff the start a is equal to the end of b .
40. $overlaps \in PER \times PER \rightarrow \mathbb{B}$: $overlaps(a, b)$ iff the end of a happens during b .
41. $overlapsInv \in PER \times PER \rightarrow \mathbb{B}$: $overlapsInv(a, b)$ iff the start of a happens during b .
42. $starts \in PER \times PER \rightarrow \mathbb{B}$: $starts(a, b)$ iff both start at the same time, but a is shorter.
43. $startsInv \in PER \times PER \rightarrow \mathbb{B}$: $startsInv(a, b)$ iff both start at the same time, but a is longer.
44. $during \in PER \times PER \rightarrow \mathbb{B}$: $during(a, b)$ iff a starts after b started and ends before b ends.
45. $duringInv \in PER \times PER \rightarrow \mathbb{B}$: $duringInv(a, b)$ iff a starts before b starts and ends after b ends.
46. $finishes \in PER \times PER \rightarrow \mathbb{B}$: $finishes(a, b)$ iff both end at the same time, but a is shorter.
47. $finishesInv \in PER \times PER \rightarrow \mathbb{B}$: $finishesInv(a, b)$ iff both end at the same time, but a is longer.
48. $matches \in PER \times PER \rightarrow \mathbb{B}$: $matches(a, b)$ iff both have the same start and end.

3.1.5 Operators on Attributes of Elements

Events, object versions, cases, and logs of the OpenSLEX meta model can be enriched with attribute values. The following functions allow the user to obtain their values:

49. $eventHasAttribute \in EVAT \times EV \rightarrow \mathbb{B}$: $true$ iff the event contains a value for a certain attribute name.
50. $versionHasAttribute \in AT \times OV \rightarrow \mathbb{B}$: $true$ iff the object version contains a value for a certain attribute name.
51. $caseHasAttribute \in CSAT \times CS \rightarrow \mathbb{B}$: $true$ iff the case contains a value for a certain attribute name.

⁵ \mathbb{B} is the set of Boolean values.

52. $\text{logHasAttribute} \in \text{LGAT} \times \text{LG} \rightarrow \mathbb{B}$: *true* iff the log contains a value for a certain attribute name.
53. $\text{getAttributeEvent} \in \text{EVAT} \times \text{EV} \not\rightarrow V$: returns the value for an attribute of an event.
54. $\text{getAttributeVersion} \in \text{AT} \times \text{OV} \not\rightarrow V$: returns the value for an attribute of an object version.
55. $\text{getAttributeCase} \in \text{CSAT} \times \text{CS} \not\rightarrow V$: returns the value for an attribute of a case.
56. $\text{getAttributeLog} \in \text{LGAT} \times \text{LG} \not\rightarrow V$: returns the value for an attribute of a log.
57. $\text{versionChange} \in \text{AT} \times V \times V \times \text{OV} \rightarrow \mathbb{B}$: *true* iff the value for an attribute linked to an object version changed from a certain value (in the previous object version) to another (in the provided object version).

By definition, getAttribute^* functions (items 53 to 56) are defined for combinations of elements and attributes for which the corresponding $^*\text{HasAttribute}$ function (items 49 to 52) evaluates to *true*.

3.1.6 Abstract Syntax

The abstract syntax of DAPOQ-Lang is defined using the notation proposed in [8]. In DAPOQ-Lang, a query is a sequence of *Assignments* combined with an *ElementSet*:

$$\begin{aligned} \text{Query} &\triangleq s : \text{Assignments}; es : \text{ElementSet} \\ \text{Assignments} &\triangleq \text{Assignment}^* \end{aligned}$$

The result of a query is an *ElementSet*, i.e., the set of elements (of the same type) from the queried OpenSLEX dataset that satisfies certain criteria. An *Assignment* assigns an *ElementSet* to a variable. Then, any reference to such variable, via its identifier, will be replaced by the corresponding *ElementSet*.

$$\begin{aligned} \text{Assignment} &\triangleq v : \text{Varname}; es : \text{ElementSet} \\ \text{Varname} &\triangleq \text{identifier} \end{aligned}$$

An *ElementSet* can be defined over other *ElementSets* by *Construction* or *Application*. It can also be defined by means of a variable identifier, i.e., an *ElementSetVar*, by a call to a terminal element function with an *ElementSetTerminal*

(Sect. 3.1.1), by computation or creation of *Periods*, or by filtering elements of the previous options with a *FilteredElementSet*.

$$\begin{aligned} \textit{ElementSet} &\triangleq \textit{Construction} \mid \textit{Application} \mid \textit{Period} \mid \textit{ElementSetVar} \mid \\ &\quad \textit{ElementSetTerminal} \mid \textit{FilteredElementSet} \\ \textit{ElementSetVar} &\triangleq \textit{identifier} \end{aligned}$$

An *ElementSetTerminal* is the *ElementSet* resulting from a call to the corresponding terminal element function (e.g., *allEvents*).

$$\begin{aligned} \textit{ElementSetTerminal} &\triangleq \textit{AllDatamodels} \mid \textit{AllClasses} \mid \textit{AllAttributes} \mid \\ &\quad \textit{AllRelationships} \mid \textit{AllObjects} \mid \textit{AllVersions} \mid \\ &\quad \textit{AllRelations} \mid \textit{AllActivityInstances} \mid \textit{AllEvents} \mid \\ &\quad \textit{AllCases} \mid \textit{AllLogs} \mid \textit{AllActivities} \mid \textit{AllProcesses} \end{aligned}$$

An *ElementSet* can be composed from other *ElementSets* by applying set operations such as *union*, *exclusion*, and *intersection*.

$$\begin{aligned} \textit{Construction} &\triangleq es_1, es_2 : \textit{ElementSet}; o : \textit{Set_Op} \\ \textit{Set_Op} &\triangleq \textit{Union} \mid \textit{Excluding} \mid \textit{Intersection} \end{aligned}$$

Also, an *ElementSet* can be constructed by means of a call to one of the *ElementOf_Op* functions, which includes the functions described in Sect. 3.1.2 that return sets of elements related to other elements, and the *periodsOf* function described in Sect. 3.1.3 that computes the periods of elements.

$$\begin{aligned} \textit{Application} &\triangleq es : \textit{ElementSet}; o : \textit{ElementOf_Op} \\ \textit{ElementOf_Op} &\triangleq \textit{datamodelsOf} \mid \textit{classesOf} \mid \textit{attributesOf} \mid \textit{relationshipsOf} \mid \\ &\quad \textit{objectsOf} \mid \textit{versionsOf} \mid \textit{relationsOf} \mid \textit{eventsOf} \mid \\ &\quad \textit{activityInstancesOf} \mid \textit{casesOf} \mid \textit{activitiesOf} \mid \textit{logsOf} \mid \\ &\quad \textit{processesOf} \mid \textit{periodsOf} \mid \textit{versionsRelatedTo} \end{aligned}$$

An *ElementSet* can be built by means of filtering, discarding elements of another *ElementSet* according to certain criteria. These criteria are expressed as a *PredicateBlock*, which will be evaluated for each member of the input *ElementSet*. Depending on the result of evaluating the *PredicateBlock*, each element will be filtered out or included in the new *ElementSet*.

$$\textit{FilteredElementSet} \triangleq es : \textit{ElementSet}; pb : \textit{PredicateBlock}$$

A *PredicateBlock* is a sequence of *Assignments* combined with a *Predicate*. Such *Predicate* can be recursively defined as a binary (*and*, *or*) or unary (*not*) combination of other *Predicates*.

$$\begin{aligned}
 \text{PredicateBlock} &\triangleq s : \text{Assignments}; p : \text{Predicate} \\
 \text{Predicate} &\triangleq \text{AttributePredicate} \mid \text{Un_Predicate} \mid \text{Bin_Predicate} \mid \\
 &\quad \text{TemporalPredicate} \\
 \text{Bin_Predicate} &\triangleq p_1, p_2 : \text{Predicate}; o : \text{BinLogical_Op} \\
 \text{Un_Predicate} &\triangleq p : \text{Predicate}; o : \text{UnLogical_Op} \\
 \text{BinLogical_Op} &\triangleq \text{And} \mid \text{Or} \\
 \text{UnLogical_Op} &\triangleq \text{Not}
 \end{aligned}$$

Also, a *Predicate* can be defined as an *AttributePredicate*, which either refers to *AttributeExists* function that checks the existence of an attribute for a certain element, an operation on attribute values (e.g., to compare attributes to substrings, constants, or other attributes), or an *AttributeChange* predicate, making use of the functions specified in Sect. 3.1.5.

$$\begin{aligned}
 \text{AttributePredicate} &\triangleq \text{AttributeExists} \mid \text{AttributeValuePred} \mid \text{AttributeChange} \\
 \text{AttributeExists} &\triangleq at : \text{AttributeName} \\
 \text{AttributeValuePred} &\triangleq at_1, at_2 : \text{Attribute}; o : \text{Value_Op} \\
 \text{AttributeChange} &\triangleq at : \text{AttributeName}; from, to : \text{Value} \\
 \text{AttributeName} &\triangleq \text{identifier} \\
 \text{Value_Op} &\triangleq == \mid >= \mid <= \mid > \mid < \mid \text{startsWith} \mid \text{endsWith} \mid \text{contains} \\
 \text{Attribute} &\triangleq \text{AttributeName} \mid \text{Value} \\
 \text{Value} &\triangleq \text{literal}
 \end{aligned}$$

Finally, a *Predicate* can be defined as a *TemporalPredicate*, i.e., a Boolean operation comparing periods or durations. *Period* comparisons based on Allen's Interval Algebra are supported by the functions defined in Sect. 3.1.4. *Duration* comparisons are done on simple scalars (e.g., ==, >, <, ≥, and ≤). A *Period* can be either created from some provided timestamps with the function *createPeriod* or computed as the global period of an element or a set of elements with the function *globalPeriodOf*. Also, a *Period* can be constructed referring to a variable containing another period by means of an identifier. *Durations* can be obtained from existing

periods (with the function *durationOf*) or created from specific durations in seconds, minutes, hours, or days with the functions defined in Sect. 3.1.3.

$$\begin{aligned}
\textit{TemporalPredicate} &\triangleq \textit{per}_1, \textit{per}_2 : \textit{Period}; \textit{o} : \textit{Period_Op} \mid \\
&\quad \textit{dur}_1, \textit{dur}_2 : \textit{Duration}; \textit{o} : \textit{Numerical_Comp_Op} \\
\textit{Period} &\triangleq \textit{PeriodCreation} \mid \textit{PeriodVar} \\
\textit{PeriodCreation} &\triangleq \textit{ts}_1, \textit{ts}_2 : \textit{Timestamp}; \textit{o} : \textit{createPeriod} \mid \\
&\quad \textit{es} : \textit{ElementSet}; \textit{o} : \textit{globalPeriodOf} \\
\textit{PeriodVar} &\triangleq \textit{identifier} \\
\textit{Period_Op} &\triangleq \textit{before} \mid \textit{after} \mid \textit{meets} \mid \textit{meetsInv} \mid \textit{overlaps} \mid \\
&\quad \textit{overlapsInv} \mid \textit{starts} \mid \textit{startsInv} \mid \textit{during} \mid \\
&\quad \textit{duringInv} \mid \textit{finishes} \mid \textit{finishesInv} \mid \textit{matches} \\
\textit{Duration} &\triangleq \textit{p} : \textit{Period}; \textit{o} : \textit{getDuration} \mid \textit{v} : \textit{Value}; \textit{o} : \textit{DurationOf} \\
\textit{DurationOf} &\triangleq \textit{Duration.ofSeconds} \mid \textit{Duration.ofMinutes} \mid \\
&\quad \textit{Duration.ofHours} \mid \textit{Duration.ofDays} \\
\textit{Numerical_Comp_Op} &\triangleq == \mid >= \mid <= \mid > \mid <
\end{aligned}$$

Figure 3 shows the syntax tree of Query 1 according to the presented abstract syntax. It also contains elements of the proposed concrete syntax, presented in detail in Sect. 5, to demonstrate the mapping to real DAPOQ-Lang queries.

3.2 Semantics

In this section, we make use of denotational semantics, as proposed in [8], to describe the meaning of DAPOQ-Lang queries. We define function M_T to describe the meaning of the nonterminal term T (e.g., M_{Query} describes the meaning of the nonterminal $Query$). First, we introduce a notation of overriding union that will be used in further discussions.

Definition 3 (Overriding Union) The overriding union of $f : X \rightarrow Y$ by $g : X \rightarrow Y$ is denoted as $f \oplus g : X \rightarrow Y$ such that $dom(f \oplus g) = dom(f) \cup dom(g)$ and

$$f \oplus g(x) = \begin{cases} g(x) & \text{if } x \in dom(g) \\ f(x) & \text{if } x \in dom(f) \setminus dom(g). \end{cases}$$

The meaning function of a query takes a query and a meta model dataset as an input and returns a set of elements that satisfy the query:

$$M_{Query} : Query \times MetaModel \rightarrow MMSets$$

This function is defined as

$$M_{Query}[q : Query, MM : MetaModel] \triangleq M_{ElementSet}(q.es, MM, M_{Assignments}(q.s, MM, \emptyset))$$

The evaluation of the query meaning function M_{Query} depends on the evaluation of the assignments and the element set involved. Evaluating the assignments means resolving their corresponding element sets and remembering the variables to which they were assigned.

$$M_{Assignments} : Assignments \times MetaModel \times Binding \rightarrow Binding$$

A sequence of assignments resolves to a binding, which links sets of elements to variable names. Assignments that happen later in the order of declaration take precedence over earlier ones when they share the variable name.

$$M_{Assignments}[s : Assignments, MM : MetaModel, B : Binding] \triangleq$$

if $\neg(s.TAIL).EMPTY$ **then**

$$M_{Assignments}(s.TAIL, MM, B \oplus M_{Assignment}(s.FIRST, MM, B))$$

else B

The result of an assignment is a binding, linking a set of elements to a variable name.

$$M_{Assignment} : Assignment \times MetaModel \times Binding \rightarrow Binding$$

$$M_{Assignment}[a : Assignment, MM : MetaModel, B : Binding] \triangleq$$

$$\{(a.v, M_{ElementSet}(a.es, MM, B))\}$$

An *ElementSet* within the context of a meta model and a binding returns a set of elements of the same type that satisfy the restrictions imposed by the *ElementSet*.

$$M_{ElementSet} : ElementSet \times MetaModel \times Binding \rightarrow MMSets$$

An *ElementSet* can be resolved as a *Construction* of other *ElementSets* with the well-known set operations of union, exclusion, and intersection. It can be the result of evaluating an *Application* function, returning elements related to other elements, the result of the creation of *Periods*, or the value of a variable previously declared

(*ElementSetVar*). Also, it can be the result of a terminal *ElementSet*, e.g., the set of all the events (*allEvents*). Finally, an *ElementSet* can be the result of filtering another *ElementSet* according to *PredicateBlock*, which is a *Predicate* preceded by a sequence of *Assignments*. These *Assignments* are only valid within the scope of the *PredicateBlock* and are not propagated outside of it (i.e., if a variable is reassigned, it will maintain its original value outside of the *PredicateBlock*). The resulting *FilteredElementSet* will contain only the elements of the input *ElementSet* for which the evaluation of the provided *Predicate* is *true*.

$M_{ElementSet}[es : ElementSet, MM : MetaModel, B : Binding] \triangleq$

case *es* **of**

Construction \Rightarrow

case *es.o* **of**

Union $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \cup M_{ElementSet}(es.es_2, MM, B)$

Excluding $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \setminus M_{ElementSet}(es.es_2, MM, B)$

Intersection $\Rightarrow M_{ElementSet}(es.es_1, MM, B) \cap M_{ElementSet}(es.es_2, MM, B)$

end

Application $\Rightarrow es.o(M_{ElementSet}(es.es, MM, B))$

Period $\Rightarrow M_{Period}(es, MM, B)$

ElementSetVar $\Rightarrow \begin{cases} B(es) & \text{if } es \in dom(B) \\ \emptyset & \text{otherwise} \end{cases}$

ElementSetTerminal $\Rightarrow es^{MM}$

FilteredElementSet $\Rightarrow \{e \in M_{ElementSet}(es.es, MM, B) \mid$

$M_{Predicate}(es.pb.p, MM, M_{Assignments}(es.pb.s, MM, B \oplus (it, e)))\}$

end

A *Predicate* is evaluated as a Boolean, with respect to a *MetaModel* and a *Binding*:

$M_{Predicate} : Predicate \times MetaModel \times Binding \rightarrow \mathbb{B}$

The meaning function of *Predicate* evaluates to a Boolean value, which can be recursively constructed combining binary (*and*, *or*) or unary (*not*) predicates. Also, a *Predicate* can be defined as an *AttributePredicate* that evaluates the existence of attributes, comparisons of attribute values, or attribute value changes. Finally, a *Predicate* can be defined as a *TemporalPredicate*, which can compare durations or periods by means of Allen's Interval Algebra operators.

$$M_{\text{Predicate}}[p : \text{Predicate}, MM : \text{MetaModel}, B : \text{Binding}] \triangleq$$

case p **of**

AttributePredicate \Rightarrow

case p **of**

AttributeExists \Rightarrow **if** $B(it) \in EV : \text{eventHasAttribute}(p.at, B(it))$

elif $B(it) \in OV : \text{versionHasAttribute}(p.at, B(it))$

elif $B(it) \in CS : \text{caseHasAttribute}(p.at, B(it))$

elif $B(it) \in LG : \text{logHasAttribute}(p.at, B(it))$

else : \emptyset

AttributeValuePred \Rightarrow

$p.o(M_{\text{Attribute}}(p.at_1, B(it), MM), M_{\text{Attribute}}(p.at_2, B(it), MM))$

AttributeChange \Rightarrow

if $B(it) \in MM.OV$ **then** : $\text{versionChange}(p.at, p.from, p.to, B(it))$ **else** : \emptyset

end

Un_Predicate $\Rightarrow \neg M_{\text{Predicate}}(p.p, MM, B)$

Bin_Predicate \Rightarrow

case $p.o$ **of**

And $\Rightarrow M_{\text{predicate}}(p.p_1, MM, B) \wedge M_{\text{predicate}}(p.p_2, MM, B)$

Or $\Rightarrow M_{\text{predicate}}(p.p_1, MM, B) \vee M_{\text{predicate}}(p.p_2, MM, B)$

end

TemporalPredicate \Rightarrow

case $p.o$ **of**

Period_Op $\Rightarrow p.o(M_{\text{Period}}(p.per_1, MM, B), M_{\text{Period}}(p.per_2, MM, B))$

Duration_Op $\Rightarrow p.o(M_{\text{Duration}}(p.dur_1, MM, B), M_{\text{Duration}}(p.dur_2, MM, B))$

end

end

A *Period* for a given meta model dataset and a binding returns an instance of *PER*, i.e., a single period element:

$$M_{\text{Period}} : \text{Period} \times \text{MetaModel} \times \text{Binding} \rightarrow \text{PER}$$

The meaning function of *Period* will return a period element that can be created (*PeriodCreation*) or assigned from a variable name containing a period (*PeriodVar*). In the case of a *PeriodCreation*, a period can be created for the specified start and end timestamps using the *createPeriod* function or it can be computed as the global period of another set of periods (*globalPeriodOf*).

$$M_{Period}[p : Period, MM : MetaModel, B : Binding] \triangleq$$

case p **of**

PeriodCreation \Rightarrow

case $p.o$ **of**

createPeriod $\Rightarrow p.o(p.ts_1, p.ts_2)$

globalPeriodOf $\Rightarrow p.o^{MM}(M_{ElementSet}(p.es, MM, B))$

end

$PeriodVar \Rightarrow \begin{cases} B(p) & \text{if } p \in dom(B) \\ \emptyset & \text{otherwise} \end{cases}$

end

A *Duration* is a value representing the length of a period, and it is computed within the context of a meta model dataset and a binding:

$$M_{Duration} : Duration \times MetaModel \times Binding \rightarrow DUR$$

A *Duration* can be evaluated based on the duration of a period (*getDuration*) or a duration specified in scalar units (*DurationOf*).

$$M_{Duration}[d : Duration, MM : MetaModel, B : Binding] \triangleq$$

case $d.o$ **of**

DurationOf $\Rightarrow d.o(d.v)$

getDuration $\Rightarrow d.o(M_{Period}(d.p, MM, B))$

end

Finally, an *Attribute* is a value assigned to an element in the context of a meta model:

$$M_{Attribute} : Attribute \times Element \times MetaModel \rightarrow Value$$

In order to evaluate the value of an *Attribute*, we can refer to the *AttributeName*, in which case the value will be obtained in different ways depending on the type

of element (event, object version, case, or log). Also, an *Attribute* can be explicitly defined by its *Value*.

$$M_{Attribute}[at : Attribute, e : Element, MM : MetaModel] \triangleq$$

case *at* **of**

AttributeName \Rightarrow

case *e* **of**

Event \Rightarrow **if** *eventHasAttribute(at, e)* **then** : *getAttributeEvent(at, e)* **else** : \emptyset

Version \Rightarrow **if** *versionHasAttribute(at, e)* **then** : *getAttributeVersion(at, e)*

else : \emptyset

Case \Rightarrow **if** *caseHasAttribute(at, e)* **then** : *getAttributeCase(at, e)* **else** : \emptyset

Log \Rightarrow **if** *logHasAttribute(at, e)* **then** : *getAttributeLog(at, e)* **else** : \emptyset

end

Value \Rightarrow *at*

end

This concludes the formal definition of DAPOQ-Lang in terms of syntax and semantics at an abstract level. The coming sections provide some details about the concrete syntax, implementation, and its performance.

4 Implementation and Evaluation

DAPOQ-Lang⁶ has been implemented as a Domain Specific Language (DSL) on top of Groovy⁷, a dynamic language for the Java platform. This means that, on top of all the functions and operators provided by DAPOQ-Lang, *any syntax allowed by Groovy or Java can be used within DAPOQ-Lang queries*. DAPOQ-Lang heavily relies on a Java implementation of the OpenSLEX⁸ meta model using SQLite⁹ as a storage and querying engine. However, DAPOQ-Lang abstracts from the specific storage choice, which allows it to run on any SQL database and not just SQLite. The platform PADAS¹⁰ (Process Aware Data Suite) integrates DAPOQ-Lang and

⁶ <https://github.com/edugonza/DAPOQ-Lang/>.

⁷ <http://groovy-lang.org/>.

⁸ <https://github.com/edugonza/OpenSLEX/>.

⁹ <https://www.sqlite.org/>.

¹⁰ <https://github.com/edugonza/PADAS/>.

Table 2 Characteristics of the three datasets employed in the evaluation

Dataset	# Objects	# Versions	# Events	# Cases	# Logs	# Activities
A	6740	8424	8512	108, 751	34	14
B	7, 339, 985	7, 340, 650	26, 106	82, 113	10, 622	172
C	162, 287	277, 094	277, 094	569, 026	29	62

OpenSLEX in a user-friendly environment to process the data and run queries. The current implementation relies on the SQLite library to store the data and execute certain subqueries. Therefore, it is to be expected that DAPOQ-Lang introduces certain overhead, given that data retrieval and object creation on the client side consume extra time and memory compared to an equivalent SQL query. In order to assess the impact of DAPOQ-Lang on query performance, we run a benchmark of pairs of equivalent queries, as expressed in DAPOQ-Lang and SQL, on the same database. The queries are organized in 3 categories and run on the 3 datasets described in [4]: A (event data obtained from the redo logs of a simulated ticket selling platform), B (event records from a financial organization), and C (ERP event data from a sample SAP system) (Table 2).

The DAPOQ-Lang queries of each pair were run with two different configurations: memory-based and disk-based caching. *Memory-based caching* uses the heap to store all the elements retrieved from the database during the execution of the query. This is good for speed when dealing with small or medium size datasets but represents a big limitation to deal with big datasets given the impact on memory use and garbage collection overhead. *Disk-based caching* makes use of MapDB,¹¹ a disk-based implementation of Java hash maps, to serialize and store on disk all the elements retrieved from the database. This significantly reduces the memory consumption and allows handling much larger datasets, which comes at the cost of speed given the overhead introduced by serialization and disk I/O operations. Figure 4 shows the results of the benchmark, with one plot per query type, one box per query engine (SQL, DAPOQ-Lang, and DAPOQ-Lang with disk caching), for the three datasets. As expected, we observe that the performance of DAPOQ-Lang queries is, in general, poorer than that of the equivalent SQL queries, especially when it comes to queries regarding the order of activities. This is due to the overhead on transmission and processing of data and the fact that many filtering operations are performed on the client instead of the server side. Obviously, there is a trade-off between ease of use and performance. Nevertheless, performance was never the main motivation for the development of DAPOQ-Lang, but ease of use and speed of query writing. In future versions, further efforts will be made to improve performance and to provide more comprehensive benchmarks.

¹¹ <http://www.mapdb.org>.

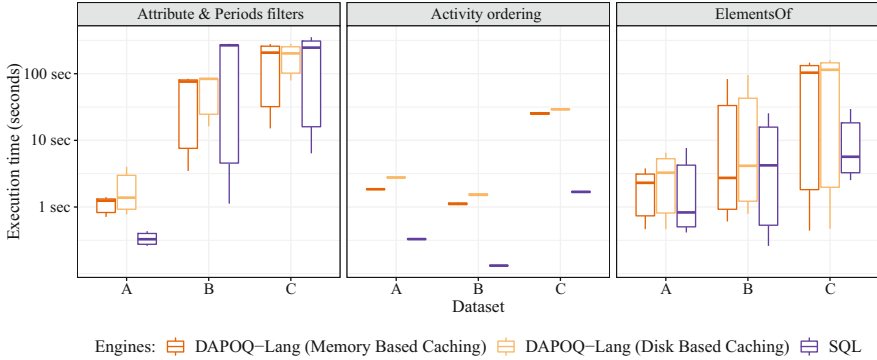


Fig. 4 Benchmark of queries run with DAPOQ-Lang, DAPOQ-Lang with disk-based caching, and SQL on an SQLite backend. Note that the vertical axis is logarithmic

5 Application and Use Cases

The purpose of this section is to demonstrate the applicability of our approach and tools. First, we explore the professional profiles, in the context of process mining, to which this language is directed to, and we identify the most common data aspects to query given each profile. Then, we provide some use cases of DAPOQ-Lang with examples of relevant queries for each data aspect. Finally, we compare DAPOQ-Lang to SQL by means of an example. The example highlights the expressiveness and compactness of our query language.

5.1 Business Questions in Process Mining

Process mining is a broad field, with many techniques available tailored toward a variety of analysis questions. “Process miners” (analysts or users carrying out a process mining project) are often interested in discovering process models from event data. Sometimes these models are provided beforehand and the focus is on conformance between the models and events. It can be the case that assessing the performance of specific activities is critical. Also, finding bottlenecks in the process can be of interest for the analysts. In some contexts, where existing regulations and guidelines impose restrictions on what is allowed and what is not in the execution of a process, compliance checking becomes a priority. In the literature, we can find examples of frequently posed questions for specific domains, like healthcare [7], in which root cause analysis becomes relevant in order to trace back data related to a problematic case. All these perspectives pose different challenges to process miners, who need to “dig” into the data to find answers to relevant questions.

Previous works [6, 9] identified professional roles and profiles in the area of business process management by analyzing job advertisements and creating a

classification based on the competencies. We make use of this classification to point out the corresponding data aspects relevant for each profile. Table 3 presents, in the two leftmost columns, the classification of the roles, as proposed by the authors of studies [6, 9]. In the column *Main Focus*, we propose, based on the role description, the sub-disciplines of process mining and data engineering that become relevant for each job profile (i.e., discovery, compliance checking, conformance checking, performance analysis, root cause analysis, integration, and data integrity). The rest of the columns indicate whether certain event data aspects become particularly interesting to be queried for each professional role, considering the role description and the main focus. We have grouped these event data aspects into two big categories that reflect the expected output of the queries: (a) *specialized sublogs* are event logs that contain only event data that reflects certain desired properties (e.g., temporal constraints, activity occurrence constraints), and (b) *metrics, artifacts, and provenance* are the resulting values of the computation of certain event data properties (e.g., performance metrics).

We see that there is a clear distinction between roles interested in performance and root cause analysis, in contrast to those mainly interested in compliance. The former will need to obtain *performance metrics* from the data, e.g., average case duration, or most time-consuming tasks. Also, they are interested in finding data related to problematic cases, e.g., obtaining all the products purchased in an unpaid order (*dependency relations*), or finding out providers of a defective batch of products (*data lineage*). However, those with a focus on compliance typically need to answer questions related to *temporal* constraints (e.g., if cases of a particular type of client are resolved within the agreed SLAs), *activity occurrence* constraints (e.g., whether a purchase was always paid), and *order of actions* (e.g., if an invoice is created before a delivery is dispatched).

As the roles get more concerned with the technical aspects of IT systems, more focus is put on performance and data properties. Especially, for technical architects, data integrity is crucial, since they are the ones in charge of integrating both applications and data storage systems. Being able to filter information based on *data properties* and find irregular *data changes* is important to verify a correct integration of different infrastructures.

Now that we have identified data aspects of interest, in what follows we present a set of example DAPOQ-Lang queries. The aim of these examples is twofold: to serve as a template to write queries and to demonstrate that the features of DAPOQ-Lang indeed cover all the aspects described in Table 3.

5.2 Exporting Logs

One of the main purposes when querying process execution data is to export it as a compatible event log format. DAPOQ-Lang provides utilities to export logs, cases, and events as XES event logs, which can be further analyzed using process mining

Table 3 Types of BPM professionals, according to [9], and relation to querying in process mining

Role [9]	Description [9]	Main focus	Specialized sublogs					Metrics, artifacts, and provenance		
			Temporal constraints	Activity occurrence	Order of actions	Data properties	Data changes	Data lineage	Dependency relations	Performance metrics
Business Process Analyst	Elicits, analyzes, documents, and communicates user requirements and designs according to business processes and IT systems; acts as a liaison between business and IT	Discovery, Compliance, and Conformance	✓	✓	✓	✓	✓			
Business Process Compliance Manager	Analyses regulatory requirements and ensures compliance of business processes and IT systems	Compliance and Conformance	✓	✓	✓	✓				
Business Process Manager, Sales and Marketing	Designs sales processes and analyses requirements for related IT systems; supports and executes sales and marketing processes	Compliance	✓	✓	✓	✓				
Business Process Improvement Manager	Analyses, measures, and continuously improves business process, e.g., through application of Lean or Six Sigma management techniques	Performance, Conformance and Root Cause Analysis	✓	✓	✓	✓	✓	✓	✓	✓
ERP Solution Architect	Implements business processes in ERP systems	Performance, Conformance and Root Cause Analysis	✓	✓	✓	✓	✓	✓	✓	✓
IT-Business Strategy Manager	Aligns business and IT strategies; monitors technological innovations and identifies business opportunities	Performance and Conformance	✓	✓	✓	✓				✓
Technical Architect	Develops and integrates hardware and software infrastructures	Integration and Data Integrity				✓	✓	✓	✓	✓

platforms such as ProM¹² or RapidProM.¹³ The following queries show the way to export XES logs for different types of data. These functions can be applied to all the query types defined under the *Specialized Sublogs* category (Table 3) in order to extract the corresponding XES event log. When a set of logs is retrieved, an independent XES log is generated for each of them.

Query 2 Export all the logs with a specific name. The result can be one or many logs being exported according to the XES format.

```
1 exportXLogsOf(allLogs().where{ name == "log01" })
```

When the input of *exportXLogsOf* is a set of cases, one single XES log is exported.

Query 3 Export in a single XES log all the cases of different logs.

```
1 exportXLogsOf(casesOf(allLogs()).where { name.contains("1" )})
```

In the case of a set of events, a single XES log with a single trace is exported (Query 4).

Query 4 Export in a single XES log all the events of different logs.

```
1 exportXLogsOf(eventsOf(allLogs()).where{ name.contains("1" )})
```

A special situation is when we want to export a set of logs or cases while filtering out events that do not comply with some criteria. In that case, we call *exportXLogsOf* with a second argument representing the set of events that can be exported (Query 5). Any event belonging to the log to be exported not contained in this set of events will be excluded from the final XES log.

Query 5 Export one or many XES logs excluding all the events that do not belong to a specific subset.

```
1 exportXLogsOf(allLogs(), eventsOf(allClasses()).where { name == "BOOKING" })
```

5.3 Specialized Sublogs

So far, we have seen how to export logs as they are stored in the dataset under analysis. However, it is very common to focus on specific aspects of the data depending on the questions to answer. This means that we need to create specialized sublogs according to certain criteria. This section presents examples of queries to

¹² <http://www.promtools.org>.

¹³ <http://www.rapidprom.org/>.

create specialized sublogs that comply with certain constraints in terms of *temporal properties*, *activity occurrence*, *order of action*, *data properties*, or *data changes*.

Temporal Constraints A way to create specialized sublogs is to filter event data based on temporal constraints. The creation and computation of periods makes it possible to select only data relevant during a certain time span. Query 6 returns events that happened during period p and that belong to the log “log01”.

Query 6 Temporal constraints. Retrieve all the events of “log01” that happened during a certain period of time.

```

1 def evLog01 = eventsOf(allLogs().where{ name == "log01" })
2 def p = createPeriod("2014/11/27_15:57", "2014/11/27_16:00", "yyyy/MM/dd_HH:mm
  ↪ ")
3
4 eventsOf(p).intersection(evLog01)

```

Query 7 focuses on the duration of cases rather than on the specific time when they happened. Only cases of log “log01” with a duration longer than 11 minutes will be returned.

Query 7 Temporal constraints. Retrieve cases of “log01” with a duration longer than 11 minutes. The variable “it” is used to iterate over all values of “c” within the “where” closure

```

1 def c = casesOf(allLogs().where{ name == "log01" })
2
3 c.where { globalPeriodOf(it).getDuration() > Duration.ofMinutes(11) }

```

Activity Occurrence Another way to select data is based on activity occurrence. The following query shows an example of how to retrieve cases in which two specific activities were performed regardless of the order. First, cases that include the first activity are retrieved (*casesA*). Then, cases that include the second activity are retrieved (*casesB*). Finally, the intersection of both sets of cases is returned.

Query 8 Activity occurrence. Retrieve cases where activities that contain the words “INSERT” or “UPDATE” and “CUSTOMER” happened in the same case.

```

1 def actA = allActivities().where {
2   name.contains("INSERT") && name.contains("CUSTOMER") }
3
4 def actB = allActivities().where {
5   name.contains("UPDATE") && name.contains("CUSTOMER") }
6
7 def casesA = casesOf(actA)
8 def casesB = casesOf(actB)
9
10 casesA.intersection(casesB)

```

Order of Actions This time we are interested in cases in which the relevant activities happened in a specific order. The following query, an extended version of Query 8, selects the cases that include both activities. Yet, before storing the intersection of cases containing events of the activities in the set *actA* with cases

containing events of the activities in the set *actB* in a variable (line 13), the query performs a filter based on the order of these two activities. To do so, for each case, the set of events is retrieved (line 15). Next, the events of the first and second activities are selected (lines 16 and 17). Finally, the periods of both events are compared (line 18), evaluating the condition to the value *true* for each case in which all the events of activity *A* happened *before* the events of activity *B*. Only the cases for which the condition block (lines 14 to 18) evaluated to *true* are stored in the variable *casesAB* and returned.

Query 9 Order of actions. Retrieve cases where activities that contain the words “INSERT” and “CUSTOMER” happen before activities that contain the words “UPDATE” and “CUSTOMER”.

```

1  def actA = allActivities().where {
2    name.contains("INSERT") && name.contains("CUSTOMER") }
3
4  def actB = allActivities().where {
5    name.contains("UPDATE") && name.contains("CUSTOMER") }
6
7  def casesA = casesOf(actA)
8  def casesB = casesOf(actB)
9
10 def eventsA = eventsOf(actA)
11 def eventsB = eventsOf(actB)
12
13 def casesAB = casesA.intersection(casesB)
14 .where {
15   def ev = eventsOf(it)
16   def evA = ev.intersection(eventsA)
17   def evB = ev.intersection(eventsB)
18   before(globalPeriodOf(evA), globalPeriodOf(evB))
19 }

```

Data Properties Some elements in our OpenSLEX dataset contain attributes that can be queried. These elements are object versions, events, cases, and logs. The following query shows how to filter events based on their attributes. First, the query compares the value of the attribute *resource* to a constant. Also, it checks if the attribute *ADDRESS* contains a certain substring. Finally, it verifies that the event contains the attribute *CONCERT_DATE*. Only events that satisfy the first and either the second or the third will be returned as a result of the query.

Query 10 Data properties. Retrieve events of resource “SAMPLE” that either have an attribute *ADDRESS* which value contains “35” or have a *CONCERT_DATE* attribute.

```

1  allEvents().where {
2    resource == "SAMPLEDB" && (at.ADDRE.contains("35") || has(at.CONCERT_DATE))}

```

Data Changes An important feature of our query language is the function named *changed*. This function determines if the value of an attribute for a certain object version changed. The function has the attribute name as a required parameter (*at:*) and two optional parameters (*from:*, and *to:*). Query 11 returns all the events related to object versions for which the value of the attribute “BOOKING_ID” changed. No restrictions are set on the specific values. Therefore, the call to *changed* will be

evaluated to *true* for an object version only if the value of the attribute in preceding version was different from the value in the current one.

Query 11 Data changes. Retrieve events that affected versions where the value of “BOOKING_ID” changed.

```
1 eventsOf(allVersions()).where { changed([at: "BOOKING_ID"]) }
```

Query 12 shows a similar example. This time we want to obtain the events related to object versions for which the attribute “SCHEDULED_DATE” changed from “11-JUN-82” to a different one.

Query 12 Data changes. Retrieve events that affected versions where the value of “SCHEDULED_DATE” changed from “11-JUN-82” to a different value.

```
1 eventsOf(allVersions()).where { changed([at: "SCHEDULED_DATE", from: "11-JUN
↪ -82"]) }
```

Query 13 instead retrieves the events related to object versions for which the attribute “SCHEDULED_DATE” changed to “22-MAY-73” from a different one.

Query 13 Data changes. Retrieve events that affected versions where the value of “SCHEDULED_DATE” changed to “22-MAY-73” from a different value.

```
1 eventsOf(allVersions()).where { changed([at: "SCHEDULED_DATE", to: "22-MAY-73"
↪ ]) }
```

Finally, Query 14 imposes a stricter restriction, retrieving only the events related to object versions for which the attribute “SCHEDULED_DATE” changed from “24-MAR-98” to “22-MAY-73”.

Query 14 Data changes. Retrieve events that affected versions where the value of “SCHEDULED_DATE” changed from “24-MAR-98” to “22-MAY-73”.

```
1 eventsOf(allVersions()).where {
2   changed([at: "SCHEDULED_DATE", from: "24-MAR-98", to: "22-MAY-73"]) }
```

5.4 Metrics, Artifacts, and Provenance

In the previous section, we have seen examples of how to obtain specialized sublogs given certain criteria. However, we do not always want to obtain events, cases, or logs as the result of our queries. In certain situations, the interest is in data objects, and their relations to other elements of the dataset, e.g., objects of a certain type, artifacts that coexisted during a given period, or data linked to other elements. Also, one can be interested in obtaining performance metrics based on existing execution data. All these elements cannot be exported as an event log, since they do not always represent event data. However, they can be linked to related events or traces. This

section shows example queries that exploit these relations and provide results that cannot be obtained as plain event logs.

Data Lineage Data lineage focuses on the lifecycle of data, its origins, and where it is used over time. DAPOQ-Lang supports data lineage mainly with the *ElementsOf* functions listed in Sect. 3.1.2. These functions return elements of a certain type linked or related to input elements of another type. As an example, we may have an interest in obtaining all the products in the database affected by a catalog update process during a certain period in which prices were wrongly set. The following query finds the cases in log “log01” whose life span overlaps with a certain period and returns the object versions related to them.

Query 15 Data lineage. Retrieves versions of objects affected by any case in “log01” whose life span overlapped with a certain period of time. The date format is specified.

```

1  def P1 = createPeriod("2014/11/27_15:56", "2014/11/27_16:30", "yyyy/MM/dd_HH:mm
2      ↪ ")
3
4  versionsOf (
5    casesOf(allLogs().where{name=="log01"})
6    .where {
7      overlaps(globalPeriodOf(it), P1)
8    }
9  )

```

Dependency Relations An important feature of the language is the ability to query existing relations between elements of different types, as well as within object versions of different classes. Query 15 showed an example of relations between elements of different types (logs to cases, cases to versions). The following query shows an example of a query on object versions related to other object versions. First, two different classes of data objects are obtained (lines 1 and 2). Then, the versions of the class “TICKET” are retrieved (line 3). Finally, the object versions related to object versions belonging to class “BOOKING” are obtained (lines 5 and 6), and only the ones belonging to class “TICKET” are selected (line 7).

Query 16 Dependency relations. Retrieve versions of ticket objects that are related to versions of booking objects.

```

1  def ticketClass = allClasses().where{ name == "TICKET" }
2  def bookingClass = allClasses().where{ name == "BOOKING" }
3  def ticketVersions = versionsOf(ticketClass)
4
5  versionsRelatedTo (
6    versionsOf(bookingClass)
7  ).intersection(ticketVersions)

```

Performance Metrics As has been previously discussed, measuring performance and obtaining metrics for specific cases or activities are very common and relevant questions for many professional roles. DAPOQ-Lang supports this aspect by computing periods and durations to measure performance. The resulting periods can be used to compute performance statistics such as average execution time or

maximum waiting time. The following query shows how to compute periods for a subset of the events in the dataset.

Query 17 Performance metrics. Retrieve periods of events belonging to activities that contain the words “UPDATE” and “CONCERT” in their name.

```

1 def actUpdateConcert = allActivities().where {
2   name.contains("UPDATE") && name.contains("CONCERT")
3 }
4
5 periodsOf(eventsOf(actUpdateConcert))

```

Query 18 demonstrates how to filter out periods based on their duration. Cases with events executed by a certain resource are selected and their periods are computed. Next, only periods with a duration longer than 11 min are returned.

Query 18 Performance metrics. Retrieve periods of a duration longer than 11 minutes computed on cases which had at least one event executed by the resource “SAMPLEDB”.

```

1 def c = casesOf(allEvents().where { resource == "SAMPLEDB" })
2
3 periodsOf(c).where { it.getDuration() > Duration.ofMinutes(11) }

```

5.5 DAPOQ-Lang vs. SQL

So far, we have seen several examples of “toy” queries to demonstrate the use of the functions and operators provided by DAPOQ-Lang. Obviously, any DAPOQ-Lang query can be computed with other Turing-complete languages. When it comes to data querying on databases, SQL is the undisputed reference. It is the common language to interact with most of the relational database implementations available today. It is a widespread language, known by many professionals from different fields. Even without considering scripting languages like PL/SQL and just with CTEs (Common Table Expressions) and Windowing, SQL has been proven to be Turing-complete [2]. Therefore, the aim of DAPOQ-Lang is not to enable new types of computations, but to ease the task of writing queries in the specific domain of process mining.

Let us consider again the generic question (GQ) presented in Sect. 1:

GQ: In which cases, there was (a) an event that happened between time T1 and T2, (b) that performed a modification in a version of class C, (c) in which the value of field F changed from X to Y?

This question involves several types of elements: cases, events, object versions, and attributes. We instantiate this query with some specific values for T1 = “1986/09/17 00:00”, T2 = “2016/11/30 19:44”, C = “CUSTOMER”, F = “ADDRESS”, X = “Fifth Avenue”, and Y = “Sunset Boulevard”. Assuming that our database already complies with the structure proposed by the OpenSLEX meta model, we can write the following SQL query to answer the question:

Query 19 Standard SQL query executed on the OpenSLEX dataset in [4] and equivalent to the DAPOQ-Lang Query 1

```

1  SELECT distinct C.id AS "id", CAT.name, CATV.value, CATV.type
2  FROM
3  "case" AS C
4  JOIN activity_instance_to_case AS AITC ON AITC.case_id = C.id
5  JOIN activity_instance AS AI ON AI.id = AITC.activity_instance_id
6  JOIN event AS E ON E.activity_instance_id = AI.id
7  JOIN event_to_object_version AS ETOV ON ETOV.event_id = E.id
8  JOIN object_version AS OV ON ETOV.object_version_id = OV.id
9  JOIN object AS O ON OV.object_id = O.id
10 JOIN class AS CL ON O.class_id = CL.id AND CL.name = "CUSTOMER"
11 JOIN attribute_name AS AT ON AT.name = "ADDRESS"
12 JOIN attribute_value AS AV ON AV.attribute_name_id = AT.id AND
13     AV.object_version_id = OV.id
14 LEFT JOIN case_attribute_value AS CATV ON CATV.case_id = C.id
15 LEFT JOIN case_attribute_name AS CAT ON CAT.id = CATV.case_attribute_name_id
16 WHERE
17     E.timestamp > "527292000000" AND
18     E.timestamp < "1480531444303" AND
19     AV.value LIKE "Sunset_Boulevard" AND
20     EXISTS
21     (
22     SELECT OVP.id
23     FROM
24     object_version AS OVP,
25     attribute_value AS AVP
26     WHERE
27     AVP.attribute_name_id = AT.id AND
28     AVP.object_version_id = OVP.id AND
29     OVP.object_id = OV.object_id AND
30     AVP.value LIKE "Fifth_Avenue" AND
31     OVP.id IN
32     (
33     SELECT OVPP.id
34     FROM object_version AS OVPP
35     WHERE
36     OVPP.end_timestamp <= OV.start_timestamp AND
37     OVPP.end_timestamp >= 0 AND
38     OVPP.object_id = OV.object_id AND
39     OVPP.id != OV.id
40     ORDER BY OVPP.end_timestamp DESC LIMIT 1
41     )
42     )

```

The logic is the following. Two subqueries are nested in order to retrieve (a) object versions preceding another object version (lines 33–40) and object versions that contain the attribute that changed (lines 22–41). Parts of the query focus on checking the value of the attributes (lines 27–30), the timestamp of the events (lines 17–18), and the class of the object versions (line 10). The rest of the query is concerned with joining rows of different tables by means of foreign keys.

The equivalent DAPOQ-Lang query, previously presented in Query 1, removes most of the clutter and boilerplate code in order to join tables together and lets the user focus on the definition of the constraints. The query is built up with an assignment and several nested queries. First, a period of time is defined (line 1). Then, object versions of a certain class are retrieved (lines 5–6) and filtered based on the changes of one of the attributes (line 7). Next, the events related to such object versions are obtained (lines 4–8) and filtered based on the time when they occurred (lines 8–12). Finally, the cases of these events are returned (lines 3–13).

Table 4 Event log obtained from the execution of Query 1

#	Case	Activity name	Timestamp	Class	Address
1	1	Insert Customer	2014-11-27 15:57:13	CUSTOMER	Fifth Avenue
2	1	Update Customer	2014-11-27 16:05:01	CUSTOMER	Sunset Boulevard
3	2	Insert Customer	2014-11-27 15:58:14	CUSTOMER	Fifth Avenue
4	2	Update Customer	2014-11-27 16:05:37	CUSTOMER	Sunset Boulevard
5	3	Insert Customer	2014-11-27 15:59:16	CUSTOMER	Fifth Avenue
6	3	Update Customer	2014-11-27 16:05:54	CUSTOMER	Sunset Boulevard
7	4	Insert Customer	2014-11-27 16:01:03	CUSTOMER	Fifth Avenue
8	4	Update Customer	2014-11-27 16:07:02	CUSTOMER	Sunset Boulevard

Table 4 shows the event log obtained from the execution of this query, where we can observe that insertions of new customers are followed by updates that modify the address attribute.

In essence, the advantage of DAPOQ-Lang over SQL is on the ease of use in the domain of process mining. The fact that we can assume how logs, cases, events, and objects are linked allows us to focus on the important parts of the query. Also, providing functions that implement the most frequent operations on data (such as period and duration computation) makes writing queries faster and less prone to errors.

6 DAPOQ-Lang and the Process Querying Framework

The Process Querying Framework (PQF) provides a comprehensive overview of the aspects involved in the process querying cycle. This framework partly originates from a collection of functional and non-functional requirements for process querying in the process management field. The requirements, based on CRUD operations (Create, Read, Update, and Delete), focus on the relevant BPM use cases presented in [11]. As has been shown in Sect. 5, our query language fulfills or supports, to some extent, the tasks involved in the requirements regarding “Check conformance using event data” and “Analyze performance using event data.” In this section, we instantiate DAPOQ-Lang in the PQF.

The first part of PQF is named “**Model, Simulate, Record, and Correlate.**” DAPOQ-Lang does not aim to support any of the aspects covered by this part of the framework. However, the OpenSLEX meta model, and more specifically its implementation, enables the *recording* and *correlation* of behavioral and historical data in a structured way, i.e., create and update operations. This feature enables the construction of a dataset ready to be queried (i.e., read operations) by DAPOQ-Lang’s query engine. Therefore, DAPOQ-Lang addresses *event log* and *correlated data* querying with the intent of retrieving information previously recorded in an OpenSLEX-compliant storage.

With respect to the “**Prepare**” part of the PQF, DAPOQ-Lang’s support is twofold: (1) It proposes the OpenSLEX meta model to structure behavioral and object data in a format that enables *indexing* and makes the querying process easier to carry out from the usability point of view. Also, it speeds up query execution providing the most frequently requested information in a preprocessed format. (2) The underlying OpenSLEX implementation makes use of *caching* to speed up response time and make efficient use of memory. Two strategies are supported: (a) in-memory caching, which benefits speed but suffers when dealing with large datasets, and (b) disk-based caching, which makes it possible to handle larger datasets that would not fit in memory but introduces an overhead due to the serialization and disk-writing steps.

As DAPOQ-Lang is a query language with an existing implementation, it covers the “**Execute**” part of the PQF. The nested nature of the expressions in DAPOQ-Lang enables the *filtering* of the data, executing parts of the query only on the relevant elements. The implementation includes several *optimizations*, like pre-fetching of attributes as a way to save time in the filtering step of the query execution. The execution of a query in DAPOQ-Lang yields results in the form of a set of elements, obtained from the original dataset, representing the subset of the original data that satisfies the expressed constraints.

Finally, the “**Interpret**” part of the framework is supported by DAPOQ-Lang in two ways: (1) enabling the *inspection* of data using explorative queries and (2) exporting the result of queries to XLog, which makes it possible to apply any existing process mining technique that requires an event log as input, while benefiting from the capabilities of DAPOQ-Lang to build relevant sublogs for the query at hand.

7 Conclusion

In the field of process mining, the need for better querying mechanisms has been identified. This work proposes a method to combine both process and data perspectives in the scope of process querying, helping with the task of obtaining insights about processes. To do so, DAPOQ-Lang, a Data-Aware Process Oriented Query Language, has been developed, which allows the analyst to select relevant parts of the data in a simple way to, among other things, generate specialized event logs to answer meaningful business questions. We have formally described the syntax and semantics of the language. We presented its application by means of simple use cases and query examples in order to show its usefulness and simplicity. In addition, we provide an efficient implementation that enables not only the execution but also the fast development of queries. This work shows that it is feasible to develop a query language that satisfies the needs of process analysts, while balancing these with demands for simplicity and ease of use. Finally, we positioned DAPOQ-Lang within the Process Querying Framework [10]. DAPOQ-Lang presents certain limitations in terms of performance, expressiveness, and ease

of use. As future work, efforts will be made on (a) expanding the language with new functionalities and constructs relevant in the process mining context, (b) improving the query planning and execution steps in order to achieve better performance, and (c) carrying out empirical evaluations with users in order to objectively assess the suitability of the language within the process mining domain.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. *Commun. ACM* **26**(11), 832–843 (1983). <https://doi.org/10.1145/182.358434>
2. Gierth, A., Fetter, D.: Cyclic tag system. In: PostgreSQL wiki (2011). <https://www.webcitation.org/6Db5tYVpi>
3. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Everything you always wanted to know about your process, but did not know how to ask. In: Dumas, M., Fantinato, M. (eds.) *Business Process Management Workshops*, pp. 296–309. Springer International Publishing, Cham (2017)
4. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Connecting databases with process mining: a meta model and toolset. *Softw. Syst. Model.* (2018). <https://doi.org/10.1007/s10270-018-0664-7>
5. IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams (2016). <https://doi.org/10.1109/IEEESTD.2016.7740858>
6. Lederer Antonucci, Y., Goeke, R.J.: Identification of appropriate responsibilities and positions for business process management success: Seeking a valid and reliable framework. *Bus. Process Manag. J.* **17**(1), 127–146 (2011)
7. Mans, R.S., van der Aalst, W.M., Vanwersch, R.J., Moleman, A.J.: Process mining in healthcare: Data challenges when answering frequently posed questions. In: *Process Support and Knowledge Representation in Health Care*, pp. 140–153. Springer (2013)
8. Meyer, B.: *Introduction to the Theory of Programming Languages*. Prentice-Hall, Upper Saddle River, NJ, USA (1990)
9. Müller, O., Schmiedel, T., Gorbacheva, E., vom Brocke, J.: Towards a typology of business process management professionals: identifying patterns of competences through latent semantic analysis. *Enterprise IS* **10**(1), 50–80 (2016). <https://doi.org/10.1080/17517575.2014.923514>
10. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.: Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.* **100**, 41–56 (2017). <https://doi.org/10.1016/j.dss.2017.04.011>. <http://www.sciencedirect.com/science/article/pii/S0167923617300787>. *Smart Business Process Management*
11. Van Der Aalst, W.M.: *Business process management: a comprehensive survey*. ISRN Softw. Eng. **2013** (2013)
12. Watson, H.J., Wixom, B.H.: The current state of business intelligence. *Computer* **40**(9), 96–99 (2007). <https://doi.org/10.1109/MC.2007.331>