OCPQ: Object-Centric Process Querying & Constraints

Aaron Küsters ✉ [0009−0006−9195−5380] and
Wil M.P. van der Aalst [0000−0002−0955−6940]

Chair of Process and Data Science (PADS), RWTH Aachen University
{kuesters,wvdaalst}@pads.rwth-aachen.de

Abstract. Process querying is used to extract information and insights from process execution data. Similarly, process constraints can be checked against input data, yielding information on which process instances violate them. Traditionally, such process mining techniques use case-centric event data as input. However, with the uptake of Object-Centric Process Mining (OCPM), existing querying and constraint checking techniques are no longer applicable. Object-Centric Event Data (OCED) removes the requirement to pick a single case notion (i.e., requiring that events belong to exactly one case) and can thus represent many real-life processes much more accurately. In this paper, we present a novel highly-expressive approach for object-centric process querying, called *OCPQ*. It supports a wide variety of applications, including OCED-based constraint checking and filtering. The visual representation of nested queries in OCPQ allows users to intuitively read and create queries and constraints. We implemented our approach using (1) a high-performance execution engine backend and (2) an easy-to-use editor frontend. Additionally, we evaluated our approach on a real-life dataset, showing the lack in expressiveness of prior work and runtime performance significantly better than the general querying solutions SQLite and Neo4j, as well as comparable to the performance-focused DuckDB.

Keywords: Object-Centric Process Mining · Querying · Constraints.

1 Introduction

In organizations, process execution data contain valuable insights that are often not leveraged to their full extent. The domain of process querying is concerned with methods and techniques for extracting such insights from event data. For example, given data of an order management process, a simple query for interesting cases could be formulated in natural language as "Find all cases where `pay order` is executed more than once". Process querying of execution data also has a strong correspondence to process constraints: Identifying violations of a process constraint, e.g., "`pay order` should be executed exactly once per case", corresponds to querying its violations (i.e., a query with the negated constraint).

To promote using this opportunity of gaining insights, it is important to allow stakeholders to query interesting scenarios in their processes themselves. For that

reason, graphical notations for constraints or queries are often introduced, as a way to also allow stakeholders without programming experience to utilize them. Similarly, there are also graphical declarative process models, like DECLARE [6], in which visual models describe an underlying set of constraint rules.

More recently, process querying and constraint approaches based on *Object-Centric Event Data* (OCED) have been proposed [2,3,4]. OCED no longer assumes a single case notion in data, i.e., that events belong to exactly one defined case. Instead, OCED contains a set of objects and a set of events of specified types. OCED also allows for relationships between objects and events, as well as between objects and objects. As such, OCED can represent many real-life processes much more accurately than traditional, flat event logs. For example, in an order management process, the different objects interacting in the process could include `customers`, `orders`, `items`, `packages`, and `employees`. Therefore, classical case-centric approaches, which assign one object per event, are too limiting [1]. These advantages of OCED translate directly to process querying techniques based on OCED. In particular, a more accurate and interconnected representation of the underlying real-life process can be queried, instead of only a flat representation that could lead to inaccurate results or misleading conclusions.

In this paper, we present an object-centric nested querying approach with an accompanying full graphical tool implementation, focusing on high expressiveness, fast runtime performance, and easy usability. An overview of the approach, including inputs and outputs, is shown in Figure 1. First, stakeholders design queries or constraints, optionally based on some regulation or specification document. The created queries can then be evaluated based on input OCED, yielding query results. The query results are visually shown in aggregation (e.g., with the total number of results or the percentage of violating instances) but can also be explored in the tool individually or exported (e.g., to a CSV or XLSX file).

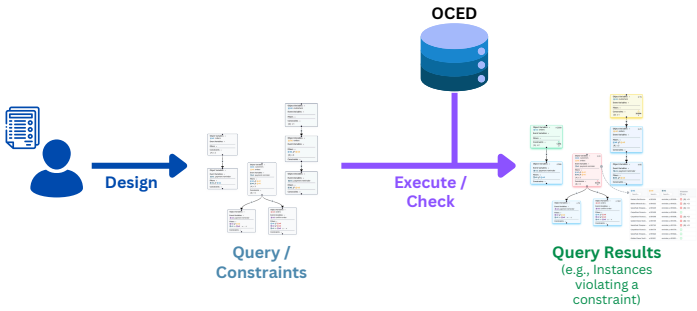


Fig. 1. Overview of the object-centric process querying and constraint approach.

Consider the following example of an object-centric constraint for an order management process: "If a third payment reminder for an order is sent to a customer, no (other) order by this customer should be confirmed afterwards".

What at first seems to be a rather simple rule actually involves multiple different object and event types, as well as multiple instances of the same object type (i.e., multiple order objects). In fact, to the best of our knowledge, none of the so far proposed graphical process querying or constraint techniques can express such a rule. Even in all-purpose querying languages, like SQL or Cypher, expressing such a rule is not only unapproachable for people without programming skills, but also largely impractical for larger data because of their poor performance for such types of queries.

We fill this gap by introducing *OCPQ*, an object-centric process querying approach, that leverages the full flexibility of the OCED data model, while focusing on easy, visual usability and a very fast runtime performance. We also present a full graphical tool implementation of the approach, which is publicly available at `https://github.com/aarkue/ocpq`. In contrast to related work, OCPQ can also express advanced queries and constraints spanning multiple object and event types while still allowing for easy visual modeling. For instance, the previously introduced example can be modeled visually as a nested query constraint in OCPQ, as shown in Figure 2.

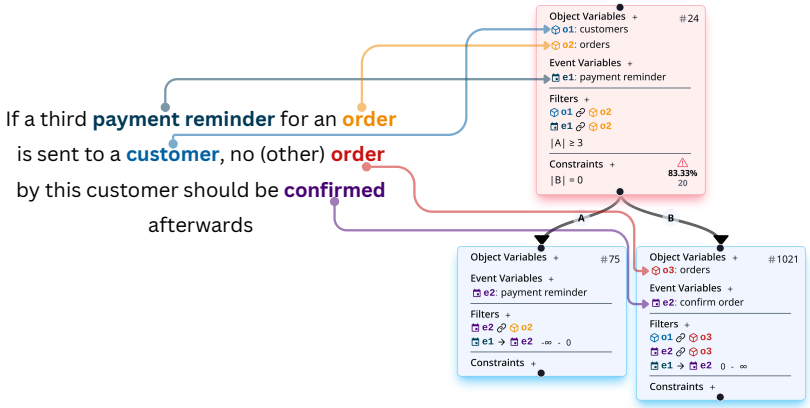


Fig. 2. The mapping of an example constraint in natural language to a visual nested query constraint in OCPQ. The included arrows indicate how objects or events mentioned in the textual description are modeled in the visual constraint.

In our tool implementation, nested queries are evaluated through a recursive, parallelizable algorithm, achieving good runtime performance and demonstrating the feasibility of our approach, even for larger, real-life datasets.

The remainder of this paper is structured as follows. In Section 2, we first discuss related work. Next, we introduce preliminaries in Section 3. Section 4 describes the main concepts of our approach. Our tool implementation is covered in Section 5, followed by an evaluation of the expressiveness and runtime in Section 6. Finally, we conclude this paper in Section 7.

2 Related Work

In this section, we present related work on process querying, process constraints and declarative process models.

Process querying research covers filtering and manipulation of process repositories, which can contain process models as well as process executions (i.e., event data), based on a (formal) query. In this paper, we focus on querying of event data without corresponding process models. In [5], the authors describe the *Process Instance Query Language* (PIQL), which allows querying the number of cases or events fulfilling specified criteria. The proprietary *Celonis Process Querying Language* (Celonis PQL) introduced in [12] is another domain-specific process querying language heavily inspired by SQL. Through the integration of Celonis PQL with the underlying data model schema, table joins do not need to be specified explicitly, as would be the case with `JOIN` statements in SQL. All the previously mentioned process querying research is primarily focused on traditional, flat event data. However, in [3], Esser et al. describe storing multi-dimensional (i.e., object-centric) event data in graph databases and using the universal graph querying language *Cypher* of the *Neo4j* database system to query entities or subgraphs.

Apart from querying, process constraints as well as declarative process models are also important related fields for this paper. Both of these fields are concerned with checking if input event data satisfies specified rules, and returning violating fragments of data if not. Most notably, *DECLARE* [6] was the first declarative approach for business process management. While internally, DECLARE uses Linear Temporal Logic (LTL), parametrized LTL templates are represented visually (e.g., as specific types of arrows) to ease usability. An initial constraint template language, also called DECLARE, is included by default, but custom template languages can be created and used as well.

In [10], Schöning et al. present an SQL-based approach for discovering declarative process constraints. By creating subqueries for returning activity combinations or computing support and confidence values, the discovery of constraints that satisfy given thresholds can be implemented as a simple SQL `SELECT` query. The authors report generally competitive performance metrics for discovering constraints. However, expanding the set of discoverable constraints, for example to ternary instead of only binary constructs, would lead to significant higher execution times. In [9], the authors extend this approach to also consider data attributes, resource, and time perspectives. Similarly, in [8], the authors present a declarative process mining framework based on SQL queries, allowing for discovery and checking of constraints. The authors additionally evaluated the time required for executing the resulting query sets across different database schemas.

There are also a few object-centric process constraint approaches. OCBC models, introduced in [2], combine behavioral constraints, inspired by DECLARE patterns, with object class data models imposing cardinality rules. As such, OCBC models can express that relationships between objects on the class-level should also manifest on the behavior-level (e.g., only items belonging to a purchase order should be associated with its pick item events). In [4], the authors

introduce *Object-Centric Constraint Graphs* (OCCGs). These constraint graphs can capture interactions between objects and events, as well as control-flow between event types, based on a given object type. Additionally, performance metrics regarding events can be included.

3 Preliminaries

As preliminaries, we first define some basic mathematical concepts.

For a set X, the *powerset* of X is $\mathcal{P}(X) = \{ Y \mid Y \subseteq X \}$. We also use partial functions: Given two sets A and B, a partial function $f \colon A \not\to B$ maps *some* of the elements of A to values in B. For elements x that are not mapped to a value (i.e., $x \notin \mathrm{dom}(f)$), we write $f(x) = \bot$. If two partial functions have disjoint domains (i.e., $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$), we write $f \cup g$ for their symmetric union. Moreover, we use subset notation (i.e., $f \subseteq g$) if it holds that $\forall_{x \in \mathrm{dom}(f)}\ g(x) = f(x)$.

Next, we define the universes that form the basis of our formalization.

Definition 1. *Let $\mathcal{U}_\Sigma$ be the universe of strings. We use the following pairwise disjoint universes:*

- $\mathcal{U}_{ev} \subseteq \mathcal{U}_\Sigma$ *Universe of events* *(e.g., e_1)*
- $\mathcal{U}_{obj} \subseteq \mathcal{U}_\Sigma$ *Universe of objects* *(e.g., o_1)*
- $\mathcal{U}_{etype} \subseteq \mathcal{U}_\Sigma$ *Universe of event types (*activities*)* *(e.g., `confirm order`)*
- $\mathcal{U}_{otype} \subseteq \mathcal{U}_\Sigma$ *Universe of object types* *(e.g., `orders`)*
- $\mathcal{U}_{attr} \subseteq \mathcal{U}_\Sigma$ *Universe of attribute names* *(e.g., `time`)*
- $\mathcal{U}_{qual} \subseteq \mathcal{U}_\Sigma$ *Universe of relationship qualifiers* *(e.g., `places`)*
- $\mathcal{U}_{\mathrm{obVar}} \subseteq \mathcal{U}_\Sigma$ *Universe of object variable names* *(e.g., `o1`)*
- $\mathcal{U}_{\mathrm{evVar}} \subseteq \mathcal{U}_\Sigma$ *Universe of event variable names* *(e.g., `e1`)*
- $\mathcal{U}_{\mathrm{setName}} \subseteq \mathcal{U}_\Sigma$ *Universe of set variable names* *(e.g., `A`)*

We write $\mathbb{T}$ for all possible timestamps and durations and $\mathcal{U}_{val}$ for the universe of all attribute values (with, for instance, $\mathbb{T} \subseteq \mathcal{U}_{val}$ and $\mathcal{U}_\Sigma \subseteq \mathcal{U}_{val}$).

Next, we introduce Object-Centric Event Data (OCED) formally. Our definition is inspired by the OCEL 2.0 specification[1], but the presented approach is not limited to any particular OCED model. At the very core, OCED contains a set of objects and a set of events, each of which have additional attributes. For objects, these attributes can also change over time. Certain types of attributes (e.g., for the object type or relationships between objects) are mandatory.

Definition 2. *Object-Centric Event Data (OCED) can be described as a tuple $L = (E, O, eaval, oaval)$ of the following components:*

- ***Events*** *$E \subseteq \mathcal{U}_{ev}$ as the set of events.*
- ***Objects*** *$O \subseteq \mathcal{U}_{obj}$ as the set of objects.*
- ***Event Attributes*** *$eaval : E \to (\mathcal{U}_{attr} \not\to \mathcal{U}_{val})$, which provides attribute values for events. For convenience, we write $eaval_e = eaval(e)$ for an $e \in E$ as a shorthand. The following properties have to hold for $eaval$:*
  - *$\forall_{e \in E}\ eaval_e(\texttt{activity}) \in \mathcal{U}_{etype}$: each event has exactly one* event type.

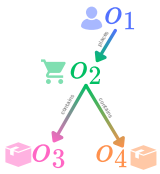
[1] `https://www.ocel-standard.org/`

- $\forall_{e \in E}\ eaval_e(\texttt{objects}) \subseteq \mathcal{U}_{qual} \times O \wedge eaval_e(\texttt{objects}) \neq \emptyset$: *each event has at least one qualified reference to an object.*
- $\forall_{e \in E}\ eaval_e(\texttt{time}) \in \mathbb{T}$: *each event has a timestamp.*

– ***Object Attributes*** $oaval : O \to (\mathcal{U}_{attr} \times \mathbb{T} \not\to \mathcal{U}_{val})$, *which provides the attribute values of an object* $o \in O$ *at a concrete timestamp. For convenience, we write* $oaval_o^t(attr) = oaval(o)(attr, t)$ *for a given* $o \in O, t \in \mathbb{T}$ *and* $attr \in \mathcal{U}_{attr}$ *as a shorthand. The following properties have to hold for* $oaval$*:*
  - *For the* time-stable *attributes* $a \in \{\texttt{objects}, \texttt{type}\} \subseteq \mathcal{U}_{attr}$*, the assigned value must not change over time. In particular, it should hold that* $\forall_{o \in O} \forall_{t \in \mathbb{T}} \forall_{t' \in \mathbb{T}}\ oaval_o^t(a) = oaval_o^{t'}(a)$*. We also write* $oaval_o(a) = oaval_o^t(a, t)$ *with an arbitrary timestamp* $t \in \mathbb{T}$ *for these attributes.*
  - $\forall_{o \in O}\ oaval_o(\texttt{type}) \in \mathcal{U}_{otype}$: *every object has exactly one* object type.
  - $\forall_{o \in O}\ oaval_o(\texttt{objects}) \subseteq \mathcal{U}_{qual} \times O$: *an object can, optionally, contain qualified references to (other) objects.*

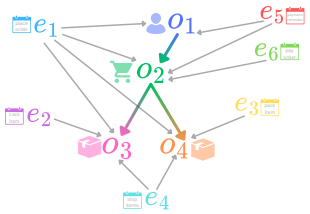
As an example OCED, consider $L = (E, O, eaval, oaval)$ with a set of objects $O = \{o_1, o_2, o_3, o_4\}$ and a set of events $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. The attribute values of all objects and events, assigned by *eaval* and *oaval*, are shown in Figure 3. For *oaval*, time-stable object attributes are marked with an $*$ in the timestamp column. Apart from the mandatory attributes, the following two custom attributes are present in L: The customer o_1 has an attribute `city`, indicating the city of residence of the customer. After providing the city initially in 2016, the attribute was updated in 2018, as the customer moved their residence. Moreover, the `payment reminder` event e_5 has an attribute `fee`, indicating the additional fine incurred by the late payment (e.g., 15€).

Object	Attribute	Timestamp	Value
o_1	type	*	customers
o_1	objects	*	{(places, o_2)}
o_1	city	2016-01-06T14:15	Bonn
o_1	city	2018-09-03T10:32	Aachen
o_2	type	*	orders
o_2	objects	*	{(contains, o_3), (contains, o_4)}
o_3	type	*	items
o_4	type	*	items



(a) *oaval* with a visualization of the O2O relationships.

Event	Attribute	Value
e_1	activity	place order
e_1	objects	{(customer, o_1), (order, o_2), (item, o_3), (item, o_4)}
e_2	activity	pack item
e_2	objects	{(item, o_3)}
e_3	activity	pack item
e_3	objects	{(item, o_4)}
e_4	activity	ship items
e_4	objects	{(ships, o_3), (ships, o_4)}
e_5	activity	payment reminder
e_5	objects	{(recipient, o_1), (order, o_2)}
e_5	fee	15
e_6	activity	pay order
e_6	objects	{(order, o_2)}



(b) *eaval* with a visualization of the O2O and E2O relationships.

Fig. 3. Example *oaval* and *eaval* tables with corresponding relationship visualizations.

The graph in Figure 3(a) shows all objects in O as nodes. Edges between nodes indicate the object-to-object (O2O) relationships, that are formally expressed through the `objects` attributes. In the graph of Figure 3(b), events and event-to-object (E2O) relationships are included additionally. For readability, the qualifiers for the O2O and E2O relationships are omitted.

For an OCED $L = (E, O, eaval, oaval)$, we introduce these shorthands:

- $E_L = E$ and $O_L = O$ for the set of objects or events of L, respectively.
- Function $type_L \in (O \cup E) \to (\mathcal{U}_{otype} \cup \mathcal{U}_{etype})$, which assigns *object or event types* to objects or events, defined as:

$$type_L(x) = \begin{cases} oaval_x(\texttt{type}), & \text{if } x \in O \\ eaval_x(\texttt{activity}), & \text{if } x \in E \end{cases}$$

- Function $time_L \in E \to \mathbb{T}$ with $time_L(e) = eaval_e(\texttt{time})$, which maps an event to its timestamp.
- Given an optional qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$, we define the function $obj_L^q \colon (E \cup O) \to \mathcal{P}(O)$, which assigns an event or object to its set of object references:

$$obj_L^q(x) = \begin{cases} \{o \mid (q', o) \in eaval_x(\texttt{objects}) \wedge (q = * \vee q = q')\}, & \text{if } x \in E \\ \{o \mid (q', o) \in oaval_x(\texttt{objects}) \wedge (q = * \vee q = q')\}, & \text{if } x \in O \end{cases}$$

  For simplicity we also write $obj_L = obj_L^*$ for all object references without considering their qualifiers.

4 Object-Centric Querying and Constraints

In this section, we detail our approach to object-centric querying and constraints. First, we introduce the concept of *variable bindings*. They make up the output of our querying approach. A variable binding is a collection of concrete objects and events of an OCED that are referred to using variable names. Through these names, events or objects can be differentiated even if they are of the same type.

Definition 3. *Let L be an OCED. The set of variable bindings $\mathbb{B}_L$ under L is:*

$$\mathbb{B}_L = \{b_1 \cup b_2 \mid b_1 \in (\mathcal{U}_{\text{evVar}} \not\to E_L) \wedge b_2 \in (\mathcal{U}_{\text{obVar}} \not\to O_L)\}$$

Consider an example OCED L with $o_1, o_2, o_3 \in O_L$ and $e_1, e_2, e_3 \in E_L$, and let $\texttt{o1}, \texttt{o2}, \texttt{o3} \in \mathcal{U}_{\text{obVar}}$ and $\texttt{e1}, \texttt{e2}, \texttt{e3} \in \mathcal{U}_{\text{evVar}}$ be object and event variable names. Then the following example bindings are part of $\mathbb{B}_L$: $b_1 = \{\}$, $b_2 = \{\texttt{o1} \mapsto o_1\}$, $b_3 = \{\texttt{o2} \mapsto o_1\}$, and $b_4 = \{\texttt{e1} \mapsto e_1, \texttt{e2} \mapsto e_3, \texttt{o1} \mapsto o_1, \texttt{o3} \mapsto o_3\}$.

In the context of an OCED, we next introduce the concept of child and parent bindings. A child binding contains all the event and object variables of the parent, and also maps them to the same values as the parent.

Definition 4. *Let L be an OCED. We define the parent-child relation $\sqsubseteq_L$ as follows: For two bindings $p, c \in \mathbb{B}_L$, $p \sqsubseteq_L c \Leftrightarrow \forall_{x \in \text{dom}(p)}\ p(x) = c(x)$. When $p \sqsubseteq_L c$, we call c a* child binding *of p and p a* parent binding *of c. Clearly, $\sqsubseteq_L$ is a partial order (i.e., reflexive, antisymmetric and transitive).*

For every OCED L, the empty binding $\{\}$ is the smallest element in $\mathbb{B}_L$ regarding $\sqsubseteq_L$. Considering the previous example bindings, it holds that $b_2 \sqsubseteq_L b_4$ and $b_3 \not\sqsubseteq_L b_4$. The $\sqsubseteq_L$ relation is useful for describing the output of *nested* queries.

Next, we introduce *binding predicates*. They allow specifying which bindings a query should return, similar to a `WHERE` clause in SQL.

Definition 5. *Let L be an OCED. Given L, a* binding predicate *describes a set of bindings that satisfy this predicate. However, different predicates can be distinguished even if they induce the same set of satisfied bindings. We write $\mathbb{P}_L$ for the set of all binding predicates. If a binding $b \in \mathbb{B}_L$ satisfies a predicate $s \in \mathbb{P}_L$, we write $b \models s$. Additionally, we use the same notation for a set of predicates $S \subseteq \mathbb{P}_L$: $b \models S \Leftrightarrow \forall_{s \in S}\ b \models s$.*

As an example binding predicate for an OCED L, consider $s \in \mathbb{P}_L$ with $s \models b \Leftrightarrow b(\texttt{o1}) \in O_L \wedge b(\texttt{e1}) \in E_L \wedge b(\texttt{o1}) \in obj_L(b(\texttt{e1}))$ for all $b \in \mathbb{B}_L$. The binding predicate s encompasses all variable bindings where the variables `o1` and `e1` are bound to objects or events of L, respectively, such that the values are in an event-to-object relationship.

One can specify (pairwise disjoint) *collections of binding predicates* for different purposes. Initially, we define the collection $\mathbf{BASIC_L} \subseteq \mathbb{P}_L$ in the context of L. Note that our approach is easily extendable by adding more predicate types and collections. However, for brevity, we do not define further predicates (e.g., based on general data attributes, like the `price` of an `orders` object) here.

$\mathbf{BASIC_L}$ is made up of three predicate types:

- **Event-to-Object** For an event variable $v \in \mathcal{U}_{\text{evVar}}$, an object variable $v' \in \mathcal{U}_{\text{obVar}}$ and an optional relationship qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$: $\text{E2O}(v, v', q) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:
$$b \models \text{E2O}(v, v', q) \ \Leftrightarrow\ b(v) \in E_L \wedge b(v') \in O_L \wedge b(v') \in obj_L^q(b(v))$$
- **Object-to-Object** For two object variables $v, v' \in \mathcal{U}_{\text{obVar}}$, and an optional qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$: $\text{O2O}(v, v', q) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:
$$b \models \text{O2O}(v, v', q) \ \Leftrightarrow\ b(v) \in O_L \wedge b(v') \in O_L \wedge b(v') \in obj_L^q(b(v))$$
- **Time Between Events** For two event variables $v, v' \in \mathcal{U}_{\text{evVar}}$ and a duration interval $t_{min}, t_{max} \in \mathbb{T}$: $\text{TBE}(v, v', t_{min}, t_{max}) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:
$$b \models \text{TBE}(v, v', t_{min}, t_{max}) \ \Leftrightarrow\ b(v) \in E_L \wedge b(v') \in E_L \wedge t_{min} \leq time_L(b(v')) - time_L(b(v)) \leq t_{max}$$

Consider the predicate $s_1 = \text{O2O}(\texttt{o1}, \texttt{o2}, *) \in \mathbf{BASIC_L}$ and the bindings $b_5 = \{\texttt{o1} \mapsto o_1\}$, and $b_6 = \{\texttt{o1} \mapsto o_1, \texttt{o2} \mapsto o_2, \texttt{e1} \mapsto e_1\}$. As b_5 does not assign `o2`, it holds that $b_5 \not\models s_1$. Assuming an object-to-object relation between o_1 and o_2 exists in L, $b_6 \models s_1$ would hold.

Next, we introduce the concept of *binding boxes*, which correspond to simple queries, yielding sets of variable bindings as output.

Definition 6. *Let L be an OCED. A binding box $\mathfrak{b}_L = (\text{Var}, \text{Pred})$ over L is a tuple consisting of:*

- $\text{Var} \in \{ev \cup ob \mid ev \in \mathcal{U}_{\text{evVar}} \not\to \mathcal{P}(\mathcal{U}_{etype}) \wedge ob \in \mathcal{U}_{\text{obVar}} \not\to \mathcal{P}(\mathcal{U}_{otype})\}$, *a partial function which specifies to values of which event or object types selected variables should be bound.*

– $\text{Pred} \subseteq \mathbb{P}_L$, *a set of binding predicates.*

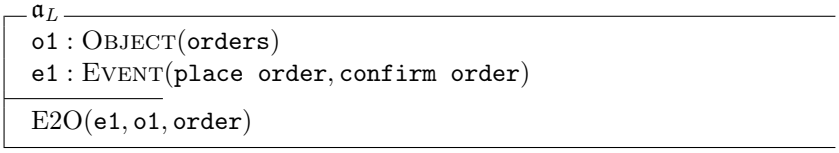
Intuitively, $\mathfrak{b}_L$ *binds the variable names* $\text{dom}(\text{Var})$ *to all combination of values (i.e., events or objects of L) such that they are of the specified types and the predicate set S holds. For convenience, we sometimes write* $\text{Var}(\mathfrak{b}_L) = \text{Var}$ *and* $\text{Pred}(\mathfrak{b}_L) = \text{Pred}$. *Additionally, we write* $\mathfrak{BOX}_L$ *for the set of all binding boxes under L. We define when a binding* $b \in \mathbb{B}_L$ *satisfies the binding box, written as* $b \models \mathfrak{b}_L$, *as follows:*

$$b \models \mathfrak{b}_L \iff b \models \text{Pred} \wedge \text{dom}(b) = \text{dom}(\text{Var}) \\ \wedge \forall_{v \in \text{dom}(\text{Var})} \left(b(v) \in E_L \cup O_L \wedge \mathit{type}_L(b(v)) \in \text{Var}(v) \right)$$

Next, we present a simple example binding box involving one event variable and one object variable. For this and further examples, consider an OCED $L = (E, O, \mathit{eaval}, \mathit{oaval})$ of an order management process that is not fully specified here for brevity. Consider the simple binding box $\mathfrak{a}_L = (\text{Var}, \text{Pred})$, with:

– $\text{Var} = \{\texttt{e1} \mapsto \{\texttt{place order}, \texttt{confirm order}\}, \texttt{o1} \mapsto \{\texttt{orders}\}\}$
– $\text{Pred} = \{\text{O2E}(\texttt{e1}, \texttt{o2}, \texttt{order})\}$

For convenience, we will use a visual notation schema for further examples: Given a binding box $\mathfrak{a}_L = (\text{Var}, \text{Pred})$, the elements of Var are split into multiple lines on the top. On the bottom (i.e., in the predicate; below the line), the filter predicates Pred are listed using their representation (e.g., E2O).



Given this example binding box $\mathfrak{a}_L$, we can also construct its output set. Assume that L contains only the objects o_1, o_2, o_3 of type `orders`, where o_1 and o_2 are associated with a `place order` event (i.e., with e_1 and e_2, respectively). Additionally, assume that o_1 is the only object that is also associated with a `confirm order` event e_3. Then we can construct the output bindings of $\mathfrak{a}_L$, $\{b \in \mathbb{B}_L \mid b \models \mathfrak{a}_L\}$, as: $\mathit{out}_L(\mathfrak{a}_L) = \big\{\{\texttt{o1} \mapsto o_1, \texttt{e1} \mapsto e_1\}, \{\texttt{o1} \mapsto o_2, \texttt{e1} \mapsto e_2\}, \{\texttt{o1} \mapsto o_1, \texttt{e1} \mapsto e_3\}\big\}$.

To facilitate nested queries, we define a relation $\preceq_L$ between binding boxes over an OCED L. It encompasses the concept of *refined* binding boxes, where new object or event variables can be introduced, and the filter is at least as strict as before.

Definition 7. *Let L be an OCED. Let* $\mathfrak{a}_L, \mathfrak{b}_L \in \mathfrak{BOX}_L$. *We say* $\mathfrak{a}_L \preceq_L \mathfrak{b}_L$ *holds if:* $\text{Var}(\mathfrak{a}_L) \subseteq \text{Var}(\mathfrak{b}_L)$ *and* $\text{Pred}(\mathfrak{a}_L) \subseteq \text{Pred}(\mathfrak{b}_L)$.

For instance, with $\mathfrak{a}_L$ from before and $\mathfrak{b}_L = (\{\texttt{o1} \to \{\texttt{orders}\}\}, \emptyset)$, it holds that $\mathfrak{b}_L \preceq_L \mathfrak{a}_L$. The concept of refined binding boxes enables nested queries, where the first binding box only queries a subset of the overall involved objects or events (e.g., only an `orders` object but no events).

Next, we define the restriction of binding boxes on only a subset of considered predicates. They enable ignoring certain predicate types for the $\preceq_L$ relation.

Definition 8. *Let L be an OCED, let $\mathfrak{a} = (\text{Var}, \text{Pred}) \in \mathfrak{BOX}_L$ be a binding box over L and let $X \subseteq \mathbb{P}_L$ be a set of binding predicates. The filter-restriction of $\mathfrak{a}$ to X, denoted as $\mathfrak{a}|_X$, is the binding box $\mathfrak{a}|_X = (\text{Var}, \text{Pred} \cap X)$ over L.*

These concepts allow defining a tree structure of binding boxes, where children are refined versions of their parents when considering only basic predicates. These *query trees* are the core of our approach and enable declarative, nested querying of objects and events.

Definition 9. *Let L be an OCED. A query tree is a tuple $T = (V, F, r, l, box)$:*

- *V is a finite set of nodes.*
- *$r \in V$ is the designated root node (i.e., the only node with no parent).*
- *$F \subseteq V \times V$ is a set of edges between nodes, such that in the directed graph (V, F) there is exactly one path from the root r to a for all $a \in V$.*
- *$l \colon F \to \mathcal{U}_{\text{setName}}$ is an injective function, assigning unique names to edges.*
- *$box \colon V \to \mathfrak{BOX}_L$ is a function which maps each node in V to a binding box over L, such that for all edges $(a, b) \in F$ with $box(a) = \mathfrak{a}$ and $box(b) = \mathfrak{b}$, it holds that $\mathfrak{a}|_{\boldsymbol{BASIC_L}} \preceq_L \mathfrak{b}|_{\boldsymbol{BASIC_L}}$.*

Next, in Figure 4, we show an example query tree only using predicates from $\mathbf{BASIC_L}$. Afterwards, we introduce more complex queries and constraint examples which motivate the specified restriction of the *box* function.

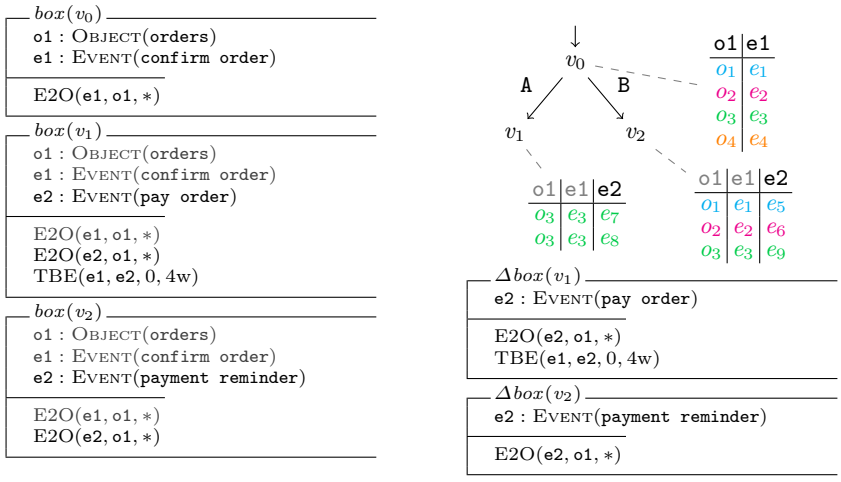


Fig. 4. A query tree with three nodes. On the left, the binding box of each node is shown. On the top right, the tree structure is visualized with example output tables. The boxes on the bottom right, marked with Δ, only show additions to their parents.

In Figure 4, the query tree $T_1 = (V, F, r, l, box)_L$ is shown, with $V = \{v_0, v_1, v_2\}$, $r = v_0$, $F = \{(v_0, v_1), (v_0, v_2)\}$, $l((v_0, v_1)) = \texttt{A}$, and $l((v_0, v_2)) = \texttt{B}$. The top right of Figure 4 shows the graph of T_1 with exemplary output binding tables, while *box* is presented on the left. Naturally, the child binding boxes contain many duplicates (shown in gray). To ease readability, we omit variables and predicates that are already present in the binding box of the parent node in future examples. With these omissions (marked using Δ), $box(v_1)$ and $box(v_2)$ can be presented more compactly, as shown on the bottom right.

On the top right of Figure 4, we show exemplary output sets of $box(v_0)$, $box(v_1)$, and $box(v_2)$ as tables next to the corresponding nodes. The rows for v_0 are colored in four different colors. For v_1 and v_2, each output row is colored based on $\sqsubseteq_L$, indicating from which parent binding in the output set of v_0 the row is derived. The first two output binding rows for v_0 (in cyan and magenta) have exactly one child binding in the output set of v_1 and none in the output set of v_2. For the third output binding row of v_0 (in green), one child binding in v_1 exists, and there are also two child bindings in the output set of v_2. The last output row of v_0 (in orange) has no child binding in the output sets of v_1 or v_2.

In a nested query, oftentimes, the result of the inner query is used in the outer query in an aggregated way. For instance, in a query for all customers with more than 100 orders, the outer query (all customers) uses the result count of the inner query (all orders by the customer) as its filter. To express such queries, we introduce a new set of binding predicates, $\textbf{CHILD SET}_u^T$, in the context of a query tree $T = (V, F, r, l, box)$ and one of its nodes $u \in V$. For every child node $v \in V$ with $(u, v) \in F$ and $l((u, v)) = A$, predicates of the form $\text{CBS}(A, n_{min}, n_{max})$ with $n_{min}, n_{max} \in \mathbb{N}_0$ are available in $\textbf{CHILD SET}_u^T$. They are fulfilled for a binding $b \in \mathbb{B}_L$, when the set of child bindings of b in v, $S = \{x \in \mathbb{B}_L \mid x \models box(v) \wedge b \sqsubseteq_L x\}$, is in the specified size range (i.e., $n_{min} \leq |S| \leq n_{max}$). Note, that these predicates can be recursive, as the predicates of a child node are already considered when evaluating the predicates of the parent node. In particular, they are also only well-defined for the binding box of a specified node, which is why they are not considered for the $\preceq_L$ relation in the tree, which only considers the predicates in $\textbf{BASIC}_\textbf{L}$.

Figure 5 shows such an extended version of the previous tree example.

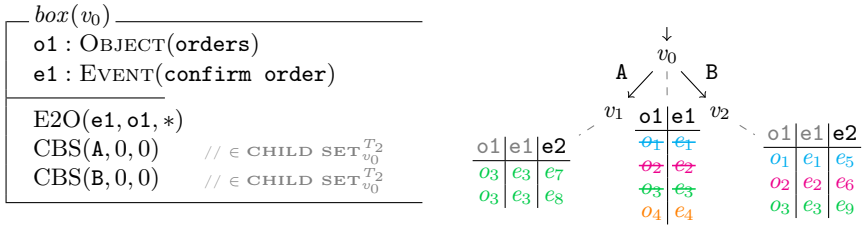


Fig. 5. An extension of the query tree from Figure 4 with child filter predicates (CBS).

Figure 5 shows a new query tree $T_2 = (V, F, r, l, box)$, with $V = \{v_0, v_1, v_2\}$, $r = v_0$, and $F = \{(v_0, v_1), (v_0, v_2)\}$. While v_1 and v_2 are the same as in Figure 4, the root node box (v_0) now contains two additional predicates of type $\mathbf{CHILD\ SET}^{T}_{v_0}$. T_2 queries placed orders that *were not paid fast* (i.e., within 4 weeks after confirmation) and for which also *no payment reminder was sent*. Again, we annotate example output tables for each node in the tree on the right of Figure 5. If bindings are removed only by a child set predicate of a binding box but fulfill the basic predicates of it (i.e., $\mathbf{BASIC_L}$), we indicate this by including this binding with a strikethrough. Generally, children of the node might still contain child bindings for crossed out parent binding rows (e.g., all rows for v_1 and v_2 in Figure 5); however, in practice, they are often not of particular interest and thus sometimes ignored or omitted from the result tables.

Next, we want to outline how the presented querying approach can be extended. In general, the output tables of the query tree nodes can be augmented and filtered freely. As augmentation, labels for each output row can be computed and added as columns (e.g., the total order volume of a customer). In the following, we will describe how *constraints* can be implemented. At its core, constraints are a special case of such general labels, defining for each output row if it should be considered *satisfied* or *violated*. As this classification is binary, a set of predicates for each node can be used for specifying the violation criteria. For a tree node $v \in V$, we write $constr(v) \subseteq \mathbb{P}_L$ for its set of constraint predicates. Fully defining these additions formally is outside the scope of this paper. However, in the following, we present a short example as a demonstration: Consider the query tree constraint $C = ((V, F, r, l, box), constr)$ shown in Figure 6, with $V = \{v_0, v_1\}$, $r = v_0$ and $F = \{(v_0, v_1)\}$. The graph (V, F) with the edge labels l is shown on the right, while box with $constr$ is presented on the left.

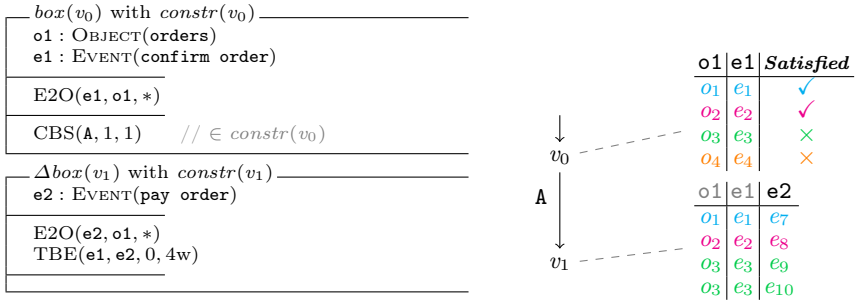


Fig. 6. Example constraint specifying that every confirmed order *should* be paid within 4 weeks after the confirmation exactly once. For each $v \in V$, the set $constr(v)$ is shown in the corresponding binding box below an extra line. In the example output tables shown on the right, this constraint is satisfied for all binding rows of $box(v_0)$ except the last two rows. The violation status of an output binding of $box(v_0)$ is annotated to the corresponding output row as either ✓ or ×.

5 Implementation

We implemented the querying and constraint checking approach introduced in Section 4 as the full-stack application *OCPQ* consisting of two parts: (1) A high-performance execution engine backend implemented in Rust and (2) an interactive user-friendly query editor frontend written in Typescript. The tool is publicly available at `https://github.com/aarkue/OCPQ`, and there are installers as well as further resources and guides available at `https://ocpq.aarkue.eu`.

Describing the implementation, and all its features, in detail is outside the scope of this paper. However, we briefly mention some key aspects:
Importing & Pre-processing: All file types of the OCEL 2.0 specification can be imported. The OCEDs are then pre-processed to enable fast query execution.
Representation of Bindings: Variable bindings are represented in a memory-efficient way, encoding the variable name and its value only through an 8-byte integer each. This enables faster execution times and usage for larger datasets.
Parallelized Recursive Query Execution Algorithm: Queries are executed by a recursive algorithm that allows for full parallelization between bindings.
HPC Deployment: To leverage this parallelization, the tool allows users to easily deploy query execution on a High-Performance Computing (HPC) cluster.
Intelligent Binding Order: New variables are bound in an efficient order, applying predicate filters as early as possible to remove unwanted bindings.

A screenshot of the tool is shown in Figure 7. The frontend editor visualizes query trees similar to the notation style that is used throughout this paper.

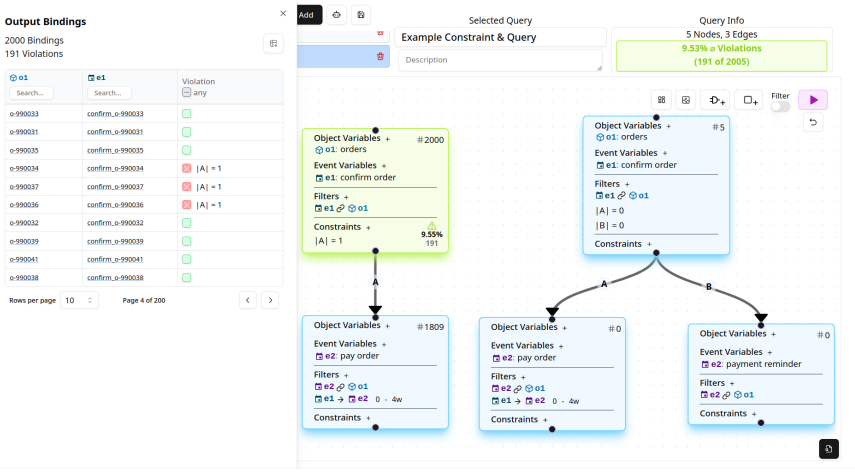


Fig. 7. Screenshot of the implemented OCPQ tool. The two shown trees correspond to the examples in this paper (i.e., Figure 6 on the left, and Figure 5 on the right). The root node of the left tree is colored according to the percentage of violations (9.55%) and its output binding results are shown in the table on the left.

6 Evaluation

For evaluation, we created seven queries and constraints for a real-life OCED dataset of a loan application process. The OCED was derived from the BPI Challenge 2017 log [11], and has more than 1,200,000 events and 100,000 objects.

The seven example queries (Q1 – Q7) are designed to be of real-life relevancy and cover a large variety of concepts, ranging from simple to more complex. In the following, we briefly introduce the used queries, first categorizing their concept or type and then describing the concrete query in natural language:

Q1 A simple constraint on the number of events that should be associated with an object: "Every application should be submitted exactly once."

Q2 A basic eventually-follows constraint (under a specified object type): "Every Offer should be returned at least once after creation."

Q3 A constraint on the number of objects of a type to be associated with events: "Each O_Returned event should involve exactly one Offer."

Q4 An eventually-follows constraint spanning across events of two related objects: "After an Application was accepted, there should be at least one associated Offer accepted afterwards."

Q5 A constraint enforcing that an object is associated with an event if it was involved with another event: "The Resource that accepts an Application should also create all Offers for that Application."

Q6 A query for the maximal duration between two events, both associated with an object: "What is the maximum delay between an Offer being created and accepted?"

Q7 A query for multiple object and event instances of the same type that are linked through another object: "Get all combinations of two Offers that are associated with the same Application and the corresponding Offer creation events."

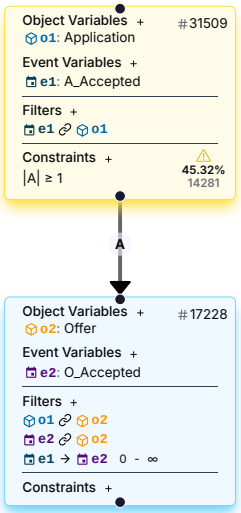
As qualitative evaluation, to investigate expressiveness, we analyzed for each query whether it (or an equivalent constraint) can be modeled in DECLARE[2] [6], OCCG [4], or OCBC [2]. As the implementations of these approaches are primarily research prototypes, and are either very slow or completely unusable for larger datasets, we did not measure their execution times. Instead, as quantitative evaluation, we compared the query execution duration of OCPQ to the general querying platforms Neo4j (used in [3]), SQLite (one of the official OCEL 2.0 formats), and the performance-focused DuckDB [7], to demonstrate the runtime performance and scalability of OCPQ.

Figure 8 shows how Q4 has been formulated in OCPQ, SQL (for SQLite and DuckDB), and Cypher (for Neo4j), as an example. While evaluating the usability of OCPQ in detail is outside the scope of this paper, this example still demonstrates the complexity of implementing simple business queries in general querying languages, like SQL or Cypher.

For SQLite, the OCEL 2.0 database was completely loaded into memory, to be a more accurate comparison to OCPQ, and it was ensured that appropriate table indices were added. For Neo4j, the database dump[3] from [3], was imported in a compatible version of Neo4j (3.5.35) and additionally Neo4j was configured to allow extensive memory usage. To measure accurate execution times for Neo4j, we used query formulations that report either the total count of results (for queries) or violations (for constraints). This was done to exclude misleading execution times for when the result rows are being streamed instead of fully computed. Moreover, these numbers are also always calculated in OCPQ.

[2] As DECLARE is based on traditional, flat event data, we assumed a reasonable flat representation of the OCED on only one object type for evaluating its expressiveness.

[3] See `https://data.4tu.nl/datasets/5c9717a0-4c22-4b78-a3ad-d2234208bfd7/1`.



(a) Q4 in OCPQ

```sql
WITH
    AcceptedOffers AS (
        SELECT DISTINCT
            O2O.ocel_source_id,
            E2.ocel_id,
            E2.ocel_time
        FROM
            object_object AS O2O
            INNER JOIN object_Offer AS A2 ON O2O.ocel_target_id = A2.ocel_id
            INNER JOIN event_object AS E2O2 ON E2O2.ocel_object_id = A2.ocel_id
            INNER JOIN event_O_Accepted AS E2 ON E2O2.ocel_event_id = E2.ocel_id
    )
SELECT
    A1.ocel_id,
    E1.ocel_id,
    (
        SELECT
            COUNT(*)
        FROM
            AcceptedOffers AS AO
        WHERE
            AO.ocel_source_id = A1.ocel_id
            AND AO.ocel_source_id = A1.ocel_id
            AND E1.ocel_time <= AO.ocel_time
    ) >= 1 AS satisfied
FROM
    object_Application AS A1
    INNER JOIN event_object AS E2O ON E2O.ocel_object_id = A1.ocel_id
    INNER JOIN event_A_Accepted AS E1 ON E2O.ocel_event_id = E1.ocel_id;
```

(b) Q4 in SQL

```cypher
MATCH (o1:Entity { EntityType: "Application" }) <-[:CORR]- (e1:Event {Activity:
↪  "A_Accepted"})
OPTIONAL MATCH (e2:Event { Activity: "O_Accepted" }) -[:CORR]-> (o2:Entity {EntityType:
↪  "Offer"}) -[:REL]-> (o1)
WHERE e1.timestamp <= e2.timestamp
WITH o1, COUNT(e2) AS e2_c, COUNT(o2) AS o2_c
WHERE e2_c < 1 OR o2_c < 1
RETURN COUNT(o1) AS violationCount
```

(c) Q4 in Cypher (violation count only)

Fig. 8. Constraint Q4 formulated in OCPQ, SQL, and Cypher.

We used the full current implementation for formulating the example queries using OCPQ, also including features that were only briefly mentioned in this paper. All queries except Q6 are modeled fully visually and without programming in OCPQ. For Q6, a scripting feature of the tool is used to calculate the maximum duration of subquery results as a label. More details on the evaluation, including the dataset, raw execution times, and formulations of the queries across all applicable approaches are available at `https://github.com/aarkue/ocpq-eval`.

The combined results for both evaluations are shown in Figure 9. The results show two things very clearly: First, there is a large gap in expressiveness, resulting in many relevant queries and constraints that are not representable in previously proposed visual approaches, like DECLARE, OCCG, or OCBC. In particular, the queries Q5 – Q7 cannot be modeled in any of them. Second, also general-purpose querying solutions, like SQLite or Neo4j, are not well suited for important types of OCED queries and are significantly slower than OCPQ for all tested queries. While DuckDB performs similar to OCPQ in terms of execution times, SQL formulations of OCED queries quickly grow complex and are difficult to write, read, and interpret. Moreover, a simple SQL translation also

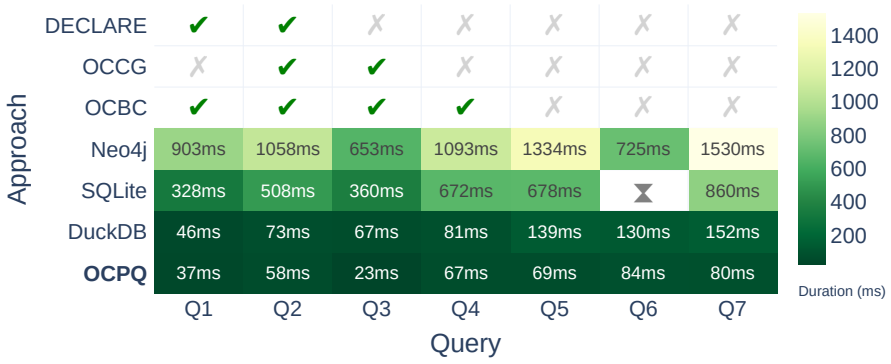


Fig. 9. Combined qualitative and quantitative evaluation of OCPQ. For the first three approaches, we only specify if the query/constraint can be expressed or not. For Neo4j, SQLite, DuckDB, and OCPQ, all queries are expressible, and we report their mean execution time across ten runs on the BPIC2017 OCED. Evaluating Q6 in SQLite took longer than four minutes and is thus omitted.

does not yield any subquery results, unlike OCPQ where results are available for each subquery.

Of course, there are some limitations to our evaluation. As we created the queries Q1 – Q7 ourselves, they might exhibit certain biases. However, they still serve as typical examples with real-world relevancy and cover a wide range of constructs. Also, while we spent special attention to crafting well-performing queries for both SQLite and Neo4j, even more efficient formulations might be possible. Finally, we performed our evaluation only on one real-life OCED dataset.

7 Conclusion

In this paper, we proposed an object-centric querying and constraint approach, *OCPQ*, based on variable bindings of objects and events. Through the use of bindings, it can express more advanced queries and constraints than previous graphical approaches. We implemented our approach as a full-stack solution, supporting efficient execution of queries and constraints, as well as an interactive editor for creating them. As evaluation, we constructed several example queries and constraints and compared OCPQ to other approaches. The evaluation demonstrated the limitations of previous work in terms of expressiveness, and showed that OCPQ also significantly outperforms several general querying solutions, namely SQLite and Neo4j, in terms of runtime.

In future work, we plan to conduct a detailed performance analysis on more datasets. Additionally, the expressiveness of the proposed approach has to be studied systematically and in more depth. Furthermore, we see a lot of potential for extensions. Our concept of object-centric querying is very universal, allowing applications for *process constraints*, *OCED filtering*, *general annotations*, as well as for generating *situation tables* as input for machine learning techniques.

Acknowledgments. The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the state government of North Rhine-Westphalia for supporting this work as part of the NHR funding.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this paper.

References

1. van der Aalst, W.M.P.: Object-Centric Process Mining: Unraveling the Fabric of Real Processes. Mathematics **11**(12) (2023)
2. van der Aalst, W.M.P., Artale, A., Montali, M., Tritini, S.: Object-Centric Behavioral Constraints: Integrating Data and Declarative Process Modelling. In: Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017. CEUR Workshop Proceedings, vol. 1879 (2017)
3. Esser, S., Fahland, D.: Multi-Dimensional Event Data in Graph Databases. Journal on Data Semantics **10**(1-2), 109–141 (2021)
4. Park, G., van der Aalst, W.M.P.: Monitoring Constraints in Business Processes Using Object-Centric Constraint Graphs. In: Process Mining Workshops - ICPM 2022 International Workshops, Bozen-Bolzano, Italy, October 23-28, 2022, Revised Selected Papers. Lecture Notes in Business Information Processing, vol. 468, pp. 479–492. Springer (2022)
5. Pérez-Álvarez, J.M., Díaz, A.C., Parody, L., Quintero, A.M.R., Gómez-López, M.T.: Process Instance Query Language and the Process Querying Framework. In: Process Querying Methods, pp. 85–111. Springer (2022)
6. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA. pp. 287–300. IEEE Computer Society (2007)
7. Raasveldt, M., Muehleisen, H.: DuckDB, `https://github.com/duckdb/duckdb`
8. Riva, F., Benvenuti, D., Maggi, F.M., Marrella, A., Montali, M.: An SQL-Based Declarative Process Mining Framework for Analyzing Process Data Stored in Relational Databases. In: Business Process Management Forum - BPM 2023 Forum, Utrecht, The Netherlands, September 11-15, 2023, Proceedings. Lecture Notes in Business Information Processing, vol. 490, pp. 214–231. Springer (2023)
9. Schönig, S., Ciccio, C.D., Maggi, F.M., Mendling, J.: Discovery of Multi-perspective Declarative Process Models. In: Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9936, pp. 87–103. Springer (2016)
10. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and Customisable Declarative Process Mining with SQL. In: Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9694, pp. 290–305. Springer (2016)
11. van Dongen, B.: BPI Challenge 2017 (Feb 2017), `https://data.4tu.nl/articles/_/12696884/1`
12. Vogelgesang, T., Ambrosy, J., Becher, D., Seilbeck, R., Geyer-Klingeberg, J., Klenk, M.: Celonis PQL: A Query Language for Process Mining. In: Process Querying Methods, pp. 377–408. Springer International Publishing, Cham (2022)