Control-Flow-Based Querying of Process Executions from Partially Ordered Event Data

Daniel Schuster[1,2][0000−0002−6512−9580], Michael Martini[1][0000−0001−5376−4424], Sebastiaan J. van Zelst[1,2][0000−0003−0415−1036], and Wil M. P. van der Aalst[1,2][0000−0002−0955−6940]

[1] Fraunhofer Institute for Applied Information Technology FIT, Sankt Augustin, Germany
{daniel.schuster,michael.martini,sebastiaan.van.zelst}@fit.fraunhofer.de
[2] RWTH Aachen University, Aachen, Germany
wvdaalst@pads.rwth-aachen.de

Abstract. Event logs, as viewed in process mining, contain event data describing the execution of operational processes. Most process mining techniques take an event log as input and generate insights about the underlying process by analyzing the data provided. Consequently, handling large volumes of event data is essential to apply process mining successfully. Traditionally, individual process executions are considered sequentially ordered process activities. However, process executions are increasingly viewed as partially ordered activities to more accurately reflect process behavior observed in reality, such as simultaneous execution of activities. Process executions comprising partially ordered activities may contain more complex activity patterns than sequence-based process executions. This paper presents a novel query language to call up process executions from event logs containing partially ordered activities. The query language allows users to specify complex ordering relations over activities, i.e., control flow constraints. Evaluating a query for a given log returns process executions satisfying the specified constraints. We demonstrate the implementation of the query language in a process mining tool and evaluate its performance on real-life event logs.

Keywords: Process Mining · Process querying · Partial orders

1 Introduction

Executing operational processes generates large amounts of event data in enterprise information systems. Analyzing these data provides great opportunities for operational improvements, for example, reduced cycle times and increased conformity with reference process models. Therefore, *process mining* [17] comprises data-driven techniques to analyze event data to gain insights into the underlying processes; for example, automatically discovered process models, conformance statistics, and performance analysis information. Since service-oriented computing is concerned with orchestrating services to form dynamic business

processes [6], process mining can provide valuable insights into the actual execution of processes within organizations [16]. These insights can then be used, for example, to define services and ultimately construct service-oriented architectures. Further, process mining provides valuable tools for service monitoring.

Most process mining techniques [17] define process executions, termed *traces*, as a sequence, i.e., a *strict total order*, of executed activities. In reality, however, processes can exhibit parallel behavior, i.e., several branches of a process are executed simultaneously. Consequently, the execution of individual activities may overlap within a single trace. Thus, traces are defined by *partially ordered* executed activities. Considering traces as partial orders, the complexity of observed control flow patterns, i.e., relations among executed activities, increases compared to sequential traces. Thus, tools are needed that facilitate the handling, filtering, and exploring of traces containing partially ordered process activities.

This paper introduces a novel query language for querying traces from an event log containing partially ordered activities. The proposed language allows the specification of six essential control flow constraints, which can be further restricted via cardinality constraints and arbitrarily combined via Boolean operators. The language design is based on standardized terms for control flow patterns in process mining. We provide a formal specification of the language's syntax and semantics to facilitate reuse in other tools. Further, we present its implementation in the process mining software tool Cortado [14], which supports partially ordered event data. Query results are visualized by Cortado using a novel trace variant visualization [13]. Finally, we evaluate the performance of the query evaluation on real-life, publicly available event logs.

The remainder of this paper is structured as follows. Sect. 2 presents related work. Sect. 3 introduces preliminaries. In Sect. 4, we introduce the proposed query language. We present an exemplary application use case of the query language in Sect. 5. In Sect. 6, we present an evaluation focusing on performance aspects of the proposed query language. Finally, Sect. 7 concludes this paper.

2 Related Work

A framework for *process querying* methods is presented in [10]. In short, process query methods differ in the input used, for instance, event logs (e.g., [3,20]) or process model repositories (e.g., [2,5]), and the goal or capabilities of the query method. Overviews of process querying languages can be found in [8,9,10,19]; the majority of existing methods focuses on querying process model repositories. Subsequently, we focus on methods that operate on event logs.

Celonis PQL [18] is a multi-purpose, textual query language that works on event logs and process models and provides a variety of query options. However, traces are considered sequentially ordered activities compared to the proposed query language in this paper. In [3], a query language is proposed that operates on a single graph, i.e., a RDF, connecting all events in an event log by user-defined correlations among events. The query language allows to partition the events by specified constraints and to query paths that start and end with events

fulfilling certain requirements. Compared to our approach, we do not initially transform the entire event log into a graph structure; instead, we operate on individual traces composed of partially ordered event data.

In [4], the authors propose a natural language interface for querying event data. Similar to [3], a graph based search is used. The approach allows specifying arbitrary queries like "Who was involved in processing case x" and "For which cases is the case attribute y greater than z." However, control flow constraints over partially ordered event data are not supported, unlike the query language proposed in this paper, which is designed exclusively for control flow constraints. In [11], the authors propose an LTL-based query language to query traces, consisting of sequentially aligned process activities, fulfilling specified constraints from an event log. In [20], the authors propose an approach to query trace fragments from various event logs that are similar to a trace fragment surrounding a selected activity from a process model using a notion of neighborhood context. Traces are, in this approach, considered sequentially ordered activities.

In summary, various process querying methods exist, most of them operating over process model repositories rather than event logs, cf. [8,9,10,19]. In short, the proposed query language differs in three main points from existing work.

1. First process querying language focusing on traces containing partially ordered activities (to the best of our knowledge)
2. Focus on traces rather than event data as a whole, i.e., executing a query returns traces satisfying the specified constraints
3. Specific focus on control flow patterns, i.e., extensive options for specifying a wide range of control flow patterns

3 Preliminaries

This section introduces notations and concepts used throughout this paper.

We denote the natural numbers by $\mathbb{N}$ and the natural numbers including 0 by $\mathbb{N}_0$. We simplify by representing timestamps by positive real numbers denoted by $\mathbb{R}^+$. We denote the universe of activity labels by $\mathcal{L}$, activity instance identifier by $\mathcal{I}^A$, and case identifier by $\mathcal{I}^C$. Further, we denote a missing value by $\perp$.

Definition 1 (Activity instances). *An activity instance* $a = (i, c, l, t_s, t_c) \in \mathcal{I}^A \times \mathcal{I}^C \times \mathcal{L} \times \left(\mathbb{R}^+ \cup \{\perp\}\right) \times \mathbb{R}^+$ *uniquely identified by* $i \in \mathcal{I}^A$ *represents the execution of an activity* $l \in \mathcal{L}$ *that was executed for the process instance identified by* $c \in \mathcal{I}^C$*. The activity instance's temporal information is given by the optional start timestamp* $t_s \in \mathbb{R}^+ \cup \{\perp\}$ *and the complete timestamp* $t_c \in \mathbb{R}^+$*. If* $t_s \neq \perp \Rightarrow t_s \leq t_c$*. We denote the universe of activity instances by* $\mathcal{A}$*.*

Let $a{=}(i, c, l, t_s, t_c) \in \mathcal{A}$ be an activity instance, we use short forms to assess the different components of a; we write a^i, a^c, a^l, a^{t_s}, and a^{t_c}.

An event log can be seen as a set of activity instances describing the same process; Table 1 shows an example. Each row corresponds to an activity instance describing the execution of an activity. For instance, the first row describes

Table 1: Example of an event log with partially ordered event data

ID			Timestamp		
Activity Instance	**Case**	**Activity Label**	**Start**	**Completion**	...
1	1	credit request received (CRR)	⊥	16.06.21 12:43:35	...
2	1	document check (DC)	17.06.21 08:32:23	18.06.21 12:01:11	...
3	1	request info. from applicant (RIP)	19.06.21 09:34:00	22.06.21 09:12:00	...
4	1	request info. from third parties (RIT)	19.06.21 14:54:00	25.06.21 08:57:12	...
5	1	document check (DC)	⊥	28.06.21 14:23:59	...
6	1	credit assessment (CA)	30.06.21 13:02:11	04.07.21 08:11:32	...
7	1	security risk assessment (SRA)	01.07.21 17:23:11	06.07.21 18:51:43	...
8	1	property inspection (PI)	⊥	05.07.21 00:00:00	...
9	1	loan-to-value ratio determined (LTV)	⊥	05.07.21 00:00:00	...
10	1	decision made (DM)	⊥	08.07.21 14:13:18	...
11	2	credit request received (CRR)	⊥	17.06.21 23:21:31	...
...	...	...	...	...	...

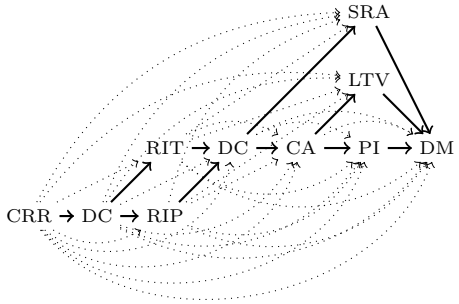


Fig. 1: Ordering of the activity instances within the trace describing case 1. Solid arcs depict the transitive reduction; solid and dotted arcs the transitive closure.

the execution of the activity "credit request received" executed on 16.06.21 at 12:43:35 for the process instance identified by case-id 1. Individual process executions within an event log are termed *traces*. Next, we formally define traces as a partially ordered set of activity instances belonging to the same case.

Definition 2 (Trace). *Let $T \subseteq \mathcal{A}$. We call $(T, \prec)$ a trace if:*

1. $\forall a_i, a_j \in T (a_i^c = a_j^c)$ *and*
2. $\prec \subseteq T \times T$ *and for arbitrary $a_i, a_j \in T$ holds that $a_i \prec a_j$ iff:*
   - $a_i^{t_c} < a_j^{t_s}$ *given that $a_i^{t_c}, a_j^{t_s} \in \mathbb{R}^+$, or*
   - $a_i^{t_c} < a_j^{t_c}$ *given that $a_i^{t_c} \in \mathbb{R}^+$ and $a_j^{t_s} = \perp$.*

We denote the universe of traces by $\mathcal{T}$.

For a trace $(T, \prec) \in \mathcal{T}$, note that the relation $\prec$ (cf. Definition 2) is the *transitive closure*. We denote the *transitive reduction* of $\prec$ by $\prec^R$. For $\prec^R$ it holds that $\forall a, b \in T \left[a \prec^R b \leftrightarrow \left(a \prec b \wedge \left(\nexists \widetilde{a} \in T (a \prec^R \widetilde{a} \wedge \widetilde{a} \prec^R b) \right) \right) \right]$. Fig. 1 visualizes the ordering relations of the activity instances of the trace describing case 1 (cf. Table 1). Solid arcs show direct relationships among activity instances. Thus, the

solid arcs represent the transitive reduction. Solid and dotted arcs represent all relations among activity instances and thus, represent the transitive closure.

Finally, we define notation conventions regarding the existential quantifier. Let $k \in \mathbb{N}$ and X be an arbitrary set, we write $\exists^{=k}$, $\exists^{\geq k}$, and $\exists^{\leq k}$ to denote that there exist *exactly*, *at least*, and *at most* k distinct elements in set X satisfying a given formula $P(\dots)$. Below we formally define the three existential quantifier.

- $\exists^{=k} x_1, \dots, x_k \in X \ \big(\forall_{1 \leq i \leq k} \ P(x_i)\big) \ \equiv \ \exists x_1, \dots, x_k \in X \big[\big(\forall_{1 \leq i < j \leq k} \ x_i \neq x_j\big) \wedge \big(\forall_{1 \leq i \leq k} \ P(x_i)\big) \wedge \big(\forall_{x \in X \setminus \{x_1, \dots, x_k\}} \neg P(x_i)\big)\big]$
- $\exists^{\geq k} x_1, \dots, x_k \in X \ \big(\forall_{1 \leq i \leq k} \ P(x_i)\big) \ \equiv \ \exists x_1, \dots, x_k \in X \big[\big(\forall_{1 \leq i < j \leq k} \ x_i \neq x_j\big) \wedge \big(\forall_{1 \leq i \leq k} \ P(x_i)\big)\big]$
- $\exists^{\leq k} x_1, \dots, x_k \in X \ \big(\forall_{1 \leq i \leq k} \ P(x_i)\big) \equiv \exists x_1, \dots, x_k \in X \big[\big(\forall_{1 \leq i \leq k} \ P(x_i)\big) \wedge \big(\forall_{x \in X \setminus \{x_1, \dots, x_k\}} \neg P(x)\big)\big]$
  Note that $x_1, \dots, x_k$ must not be *different* elements in the formula above; it specifies that at most k distinct elements in X exist satisfying $P(\dots)$.

4 Query Language

This section introduces the proposed query language. Sect. 4.1 introduces its syntax, while Sect. 4.2 defines its semantics. Sect. 4.3 covers the evaluation of queries. Finally, Sect. 4.4 presents the implementation in a process mining tool.

4.1 Syntax

This section introduces the syntax of the proposed query language. In total, six operators exist, allowing to specify control flow constraints. Table 2 provides an overview of these six operators, three binary, (i.e., `isContained` (`isC`), `isStart` (`isS`), and `isEnd` (`isE`)), and three unary operators (i.e., `isDirectlyFollowed` (`isDF`), `isEventuallyFollowed` (`isEF`), and `isParallel` (`isP`)). Next to each operator, we list query examples, including the corresponding operator, and present its semantics in natural language. As the examples show, each operator can be additionally constrained by a cardinality. We call a query a leaf query if only one operator is used, for instance, all examples shown in Table 2 are query leaves. Query leaves can be arbitrarily combined via Boolean operators, for instance, see Fig. 2. Next, we formally define the query language's syntax.

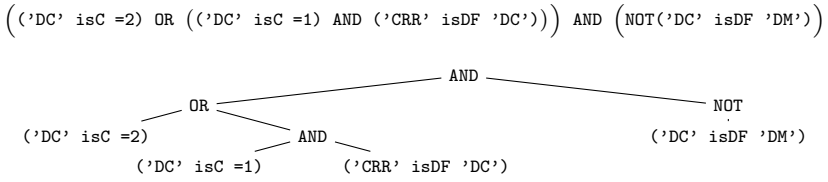


Fig. 2: Example of a query. Leaves represent individual control flow constraints (cf. Table 2) that are combined via Boolean operators.

Table 2: Overview of the six control flow constraints and corresponding examples

Type	Syntax	Example Nr.	Query	Description of semantics
unary	isContained (isC)	E1	`'A' isC`	activity `A` is contained in the trace
		E2	`'A' isC` ≥ 6	activity `A` is contained at least 6 times in the trace
		E3	`ALL{'A','B'} isS` ≥ 6	activity `A` and `B` are both contained at least 6 times each in the trace
	isStart (isS)	E4	`'A' isS`	there exists a start activity `A`[(a)]
		E5	`'A' isS = 1`	exactly one start activity of the trace is an `A` activity[(a)]
		E6	`ANY{'A','B'} isC = 1`	trace starts with exactly one `A` activity or/and with exactly one `B` activity[(a)]
	isEnd (isE)	E7	`'A' isE`	there exists an end activity `A`[(a)]
		E8	`'A' isE` ≥ 2	at least two end activities of the trace are an `A` activity[(a)]
		E9	`ALL{'A','B'} isE`	trace ends with at least one `A` and one `B` activity[(a)]
binary	isDirectly Followed (isDF)	E10	`'A' isDF 'B'`	a `B` activity directly follows *each* `A` activity in the trace
		E11	`'A' isDF 'B' = 1`	trace contains exactly one `A` activity that is directly followed by `B`
		E12	`'A' isDF ALL{'B','C'}`	every `A` activity is directly followed by a `B` and `C` activity
	isEventually Followed (isEF)	E13	`'A' isEF 'B'`	after *each* `A` activity in the trace a `B` activity eventually follows
		E14	`'A' isEF 'B'` ≥ 1	trace contains at least one `A` activity that is eventually followed by `B`
		E15	`ALL{'A','B'} isEF 'C'`	all `A` and `B` activities are eventually followed by a `C` activity
	isParallel (isP)	E16	`'A' isP 'B'`	each `A` activity in the trace is in parallel to some `B` activity
		E17	`'A' isP 'B'` ≤ 4	trace contains at most four `A` activities that are in parallel to some `B` activity
		E18	`'A' isP ANY{'B','C'}` ≤ 2	trace contains at most two `A` activities that are parallel to a `B` or `C` activity

[(a)] Trace may contain arbitrary further start respectively end activities.

Definition 3 (Query Syntax). *Let* $l_1, \ldots, l_{n-1}, l_n \in \mathcal{L}$ *be activity labels,* $k \in \mathbb{N}_0$, $\square \in \{\leq, \geq, =\}$, $\circ \in \{\mathit{isDF}, \mathit{isEF}, \mathit{isP}\}$, $\bullet \in \{\mathit{isC}, \mathit{isS}, \mathit{isE}\}$, *and* $\triangle \in \{\mathit{ALL}, \mathit{ANY}\}$. *We denote the universe of queries by* $\mathcal{Q}$ *and recursively define a query* $Q \in \mathcal{Q}$ *below.*

Leaf query with an unary operator *(without/with cardinality constraint)*

- $Q = \text{'}l_1\text{'} \bullet$ $\quad$ $Q = \text{'}l_1\text{'} \bullet \square k$
- $Q = \triangle\{\text{'}l_1\text{'}, \ldots, \text{'}l_{n-1}\text{'}\} \bullet$ $\quad$ $Q = \triangle\{\text{'}l_1\text{'}, \ldots, \text{'}l_{n-1}\text{'}\} \bullet \square k$

Leaf query with a binary operator *(without/with cardinality constraint)*

- $Q{=}\,'l_1\,' \circ\, 'l_n\,'$ $\quad$ $Q{=}\,'l_1\,' \circ\, 'l_n\,' \,\square k$
- $Q{=}\triangle\{\,'l_1\,', \ldots, 'l_{n-1}\}\,' \circ\, 'l_n\,'$ $\quad$ $Q{=}\triangle\{\,'l_1\,', \ldots, 'l_{n-1}\}\,' \circ\, 'l_n\,' \,\square k$
- $Q{=}\,'l_n\,' \circ \triangle\{\,'l_1\,', \ldots, 'l_{n-1}\}\,'$ $\quad$ $Q{=}\,'l_n\,' \circ \triangle\{\,'l_1\,', \ldots, 'l_{n-1}\}\,' \,\square k$

Composed query using Boolean operators

- *If* $Q_1, Q_2 {\in} \mathcal{Q}$ *are two queries and* $\blacksquare {\in} \{\texttt{AND}, \texttt{OR}\}$*, then* $Q{=}(Q_1 \blacksquare Q_2)$ *is a query*
- *If* $Q_1 {\in} \mathcal{Q}$ *is a query, then* $Q{=}\texttt{NOT}(Q_1)$ *is a query*

4.2 Semantics

This section introduces the query language's semantics. Table 2 presents query examples with corresponding semantics. In short, the unary operators allow to specify the existence of individual activities within a trace, for example, is contained (`isC`), is a start activity (`isS`), or is an end activity (`isE`). Optionally, operators can have cardinality constraints that extend the existential semantics of unary operators by quantification constraints. Binary operators allow to specify relationships between activities; for example, two activities are parallel (`isP`), directly follow each other (`isDF`), or eventually follow each other (`isEF`). In contrast to unary operators, binary operators always have to hold globally when no cardinality constraint is given. For example, E10 (cf. Table 2) specifies that a B activity must directly follow each A activity, i.e., there is an arc in the transitive reduction from each A activity to a B activity. In comparison, E11 specifies that the trace contains precisely one A activity that is directly followed by a B activity. `ALL` sets specify that a constraint must be fulfilled for all activity labels within the set. Analogously, `ANY` sets specify that the constraint must be fulfilled at least for one activity. Next, we formally define the semantics.

Definition 4 (Query Semantics). *Let* $Q, Q_1, Q_2 {\in} \mathcal{Q}$ *be queries,* $T^{\prec}{=}(T, \prec) \in \mathcal{T}$ *be a trace, and* $l_1, \ldots, l_n {\in} \mathcal{L}$ *be activity labels. We recursively define the function* $eval : \mathcal{Q}{\times}\mathcal{T} \to \{true, false\}$ *assigning a Boolean value, i.e.,* $eval(Q, T^{\prec})$*, to query* Q *and trace* $T^{\prec}$*.*

Unary operators:

- *If* $\boldsymbol{Q{=}\,'l_1\,'\ \texttt{isC}\ \square k}$*,* $eval\big(Q, T^{\prec}\big) \Leftrightarrow$
  $\exists^{\square k} a_1, \ldots, a_k {\in} T \big[\forall_{1 \le i \le k} (a_i^l {=} l_1)\big]$
- *If* $\boldsymbol{Q{=}\,'l_1\,'\ \texttt{isS}\ \square k}$*,* $eval\big(Q, T^{\prec}\big) \Leftrightarrow$
  $\exists^{\square k} a_1, \ldots, a_k {\in} T \Big[\forall_{1 \le i \le k} \Big(a_i^l {=} l_1 \wedge \neg \exists \widetilde{a} {\in} T (\widetilde{a} {\prec} a_i)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1\,'\ \texttt{isE}\ \square k}$*,* $eval\big(Q, T^{\prec}\big) \Leftrightarrow$
  $\exists^{\square k} a_1, \ldots, a_k {\in} T \Big[\forall_{1 \le i \le k} \Big(a_i^l {=} l_1 \wedge \neg \exists \widetilde{a} {\in} T (a_i {\prec} \widetilde{a})\Big)\Big]$

Binary operators:

- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isDF}\ 'l_2'}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a{\prec}^R\widetilde{a}\right)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isDF}\ 'l_2'\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a_i{\prec}^R\widetilde{a}\right)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isDF ANY}\{\,'l_2',\ldots,'l_n'\}}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\Big(a{\prec}^R\widetilde{a} \wedge \big(\bigvee_{j=2}^n \widetilde{a}^l = l_j\big)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isDF ANY}\{\,'l_2',\ldots,'l_n'\}\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\big(a_i{\prec}^R\widetilde{a} \wedge \bigvee_{j=2}^n(\widetilde{a}^l{=}l_j)\big)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isDF ALL}\{\,'l_2',\ldots,'l_n'\}\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}_2,\ldots,\widetilde{a}_n{\in}T\big(\bigwedge_{j=2}^n\left(a_i{\prec}^R\widetilde{a}_j \wedge \widetilde{a}_j^l{=}l_j\right)\big)\Big)\Big]$

- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isEF}\ 'l_2'}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a{\prec}\widetilde{a}\right)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{iEF}\ 'l_2'\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a_i{\prec}\widetilde{a}\right)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isEF ANY}\{\,'l_2',\ldots,'l_n'\}}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\Big(a{\prec}\widetilde{a} \wedge \big(\bigvee_{i=2}^n \widetilde{a}^l = l_i\big)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isEF ANY}\{\,'l_2',\ldots,'l_n'\}\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\big(a_i{\prec}\widetilde{a} \wedge \big(\bigvee_{j=2}^n \widetilde{a}^l{=}l_j\big)\big)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isEF ALL}\{\,'l_2',\ldots,'l_n'\}\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}_2,\ldots,\widetilde{a}_n{\in}T\big(\bigwedge_{j=2}^n(a_i{\prec}\widetilde{a}_j \wedge \widetilde{a}_j^l{=}l_j)\big)\Big)\Big]$

- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isP}\ 'l_2'}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a{\not\prec}\widetilde{a} \wedge \widetilde{a}{\not\prec}a\right)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isP}\ 'l_2'\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\left(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\left(\widetilde{a}^l{=}l_2 \wedge a_i{\not\prec}\widetilde{a} \wedge \widetilde{a}{\not\prec}a_i\right)\right)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isP ANY}\{\,'l_2',\ldots,'l_n'\}}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\forall a{\in}T\Big[a^l{=}l_1 \rightarrow \exists\widetilde{a}{\in}T\Big(a{\not\prec}\widetilde{a} \wedge \widetilde{a}{\not\prec}a \wedge \big(\bigvee_{j=2}^n \widetilde{a}^l = l_j\big)\Big)\Big]$
- *If* $\boldsymbol{Q{=}\,'l_1'\ \texttt{isP ANY}\{\,'l_2',\ldots,'l_n'\}\ \Box k}$, *then* $eval(Q,T^{\prec}) \Leftrightarrow$
  $\exists^{\Box k}a_1,\ldots,a_k{\in}T\Big[\forall_{1\leq i\leq k}\Big(a_i^l{=}l_1 \wedge \exists\widetilde{a}{\in}T\big(a_i{\not\prec}\widetilde{a} \wedge \widetilde{a}{\not\prec}a_i \wedge \big(\bigvee_{j=2}^n \widetilde{a}^l{=}l_j\big)\big)\Big)\Big]$

– *If* $\boldsymbol{Q}$=`'`$\boldsymbol{l_1}$`' isP ALL{'`$\boldsymbol{l_2}$`',...,'`$\boldsymbol{l_n}$`'}` $\boldsymbol{\square k}$, *then* $eval(Q, T^{\prec}) \Leftrightarrow$
$\exists^{\square k} a_1, \ldots, a_k \in T \Big[\forall_{1 \leq i \leq k} \Big(a_i^l = l_1 \wedge \exists \widetilde{a}_2, \ldots, \widetilde{a}_n \in T\big(\bigwedge_{j=2}^{n}(a_i \not\prec \widetilde{a}_j \wedge \widetilde{a}_j \not\prec a_i \wedge \widetilde{a}_j^l = l_j)\big)\Big)\Big]$

Boolean operators:

– *If* $\boldsymbol{Q}$=`NOT(`$\boldsymbol{Q_1}$`)`, *then* $eval(Q, T^{\prec}) \Leftrightarrow \neg eval(Q_1, T^{\prec})$
– *If* $\boldsymbol{Q}$=`(`$\boldsymbol{Q_1}$` OR `$\boldsymbol{Q_2}$`)`, *then* $eval(Q, T^{\prec}) \Leftrightarrow eval(Q_1, T^{\prec}) \vee eval(Q_2, T^{\prec})$
– *If* $\boldsymbol{Q}$=`(`$\boldsymbol{Q_1}$` AND `$\boldsymbol{Q_2}$`)`, *then* $eval(Q, T^{\prec}) \Leftrightarrow eval(Q_1, T^{\prec}) \wedge eval(Q_2, T^{\prec})$

Note that Definition 4 does not cover all queries constructible using the syntax in Definition 3. However, any query can be rewritten into a *logically equivalent* one covered by Definition 3. We call queries $Q_1, Q_2 \in \mathcal{Q}$ logically equivalent, denoted $Q_1 \equiv Q_2$, iff $\forall T^{\prec} \in \mathcal{A}^*\big(eval(Q_1, T^{\prec}) \Leftrightarrow eval(Q_2, T^{\prec})\big)$. Below, we list query rewriting rules.

– $'l_1'\bullet \equiv 'l_1'\bullet \geq 1$
– `ANY{'`l_1`',...,'`l_n`'}`$\bullet \equiv$ `('`l_1`'`$\bullet$`) OR...OR ('`l_n`'`$\bullet$`)`
– `ALL{'`l_1`',...,'`l_n`'}`$\bullet \equiv$ `('`l_1`'`$\bullet$`) AND...AND ('`l_n`'`$\bullet$`)`
– `ANY{'`l_1`',...,'`l_n`'}`$\bullet\,\square$`k` $\equiv$ `('`l_1`'`$\bullet\square$`k) OR...OR ('`l_n`'`$\bullet\,\square$`k)`
– `ALL{'`l_1`',...,'`l_n`'}`$\bullet\,\square$`k` $\equiv$ `('`l_1`'`$\bullet\,\square$`k) AND...AND ('`l_n`'`$\bullet\,\square$`k)`
– `ANY{'`l_1`',...,'`l_{n-1}`'}`$\circ$`'`l_n`'` $\equiv$ `('`l_1`'`$\circ$`'`l_n`') OR...OR ('`l_{n-1}`'`$\circ$`'`l_n`')`
– `ALL{'`l_1`',...,'`l_{n-1}`'}`$\circ$`'`l_n`'` $\equiv$ `('`l_1`'`$\circ$`'`l_n`') AND...AND ('`l_{n-1}`'`$\circ$`'`l_n`')`
– `ANY{'`l_1`',...,'`l_{n-1}`'}`$\circ$`'`l_n`'`$\square$`k` $\equiv$ `('`l_1`'`$\circ$`'`l_n`'`$\square$`k) OR...OR ('`l_{n-1}`'`$\circ$`'`l_n`'`$\square$`k)`
– `ALL{'`l_1`',...,'`l_{n-1}`'}`$\circ$`'`l_n`'`$\square$`k` $\equiv$ `('`l_1`'`$\circ$`'`l_n`'`$\square$`k) AND ... AND ('`l_{n-1}`'`$\circ$`'`l_n`'`$\square$`k)`
– $'l_1'\circ$`ALL{'`l_2`',...,'`l_n`'}` $\equiv$ `('`l_1`'`$\circ$`'`l_2`') AND...AND ('`l_1`'`$\circ$`'`l_n`')`

Note that according to Definition 4, the following queries are *not* logically equivalent. Thus, `ANY` and `ALL` sets are not syntactic sugar.

– $'l_1'\circ$`ANY{'`l_2`',...,'`l_n`'}` $\not\equiv$ `('`l_1`'`$\circ$`'`l_2`') OR...OR ('`l_1`'`$\circ$`'`l_n`')`
– $'l_1'\circ$`ANY{'`l_2`',...,'`l_n`'}`$\,\square$`k` $\not\equiv$ `('`l_1`'`$\circ$`'`l_2`'`$\square$`k) OR...OR ('`l_1`'`$\circ$`'`l_n`'`$\,\square$`k)`
– $'l_1'\circ$`ALL{'`l_2`',...,'`l_n`'}`$\,\square$`k` $\not\equiv$ `('`l_1`'`$\circ$`'`l_2`'`$\,\square$`k) AND...AND ('`l_1`'`$\circ$`'`l_n`'`$\,\square$`k)`

For example, consider E18 in Table 2. The query states that there exist at most two A activities that are in parallel to B or C activities. Thus, a trace containing four A activities, two parallel to an arbitrary number (greater than zero) of B activities, and two parallel to C activities, does not fulfill query E18. However, the described trace fulfills the query $Q =$ `('A' isP 'B'` ≤ 2`) OR ('A' isP 'C'` ≤ 2`)`; hence, $E18 =$ `'A' isP ANY{'B','C'}` $\leq 2 \not\equiv Q$.

4.3 Evaluating Queries

This section briefly discusses our approach to query evaluation. As shown in Fig. 2, queries represent trees. Since each leaf represents a query, we evaluate the queries composed of Boolean operators bottom-up. First, the leaves are evaluated

on a given trace, resulting in Boolean values per leaf. Then, bottom-up, the given Boolean operators are applied recursively.

In many cases, however, a complete query evaluation is not needed to determine its overall Boolean value for a given trace. For instance, if one leaf of a logical `AND` parent evaluates to false, the other leaves do not need to be further evaluated for the given trace. Similar applies to the logical `OR`. Reconsider the query given in Fig. 2 and the trace depicted in Fig. 1. The query consists of four leaves; however, only two must be evaluated. Following a depth-first traversing strategy, we first evaluate the leaf `('DC' isC =2)` satisfied by the given trace. Thus, we do not need to evaluate the right subtree of the `OR`, i.e., leaves `('DC' isC =1)` and `('CRR' isDF 'DC')`. Finally, we evaluate the leave `('DC' isDF 'DM')`. In short, by evaluating only two leaves, we can evaluate the entire query.

4.4 Implementation

This section briefly demonstrates the implementation of the proposed query language in the process mining tool Cortado [14][3]. We refer to [14] for an introduction to Cortado's architecture and a feature overview.

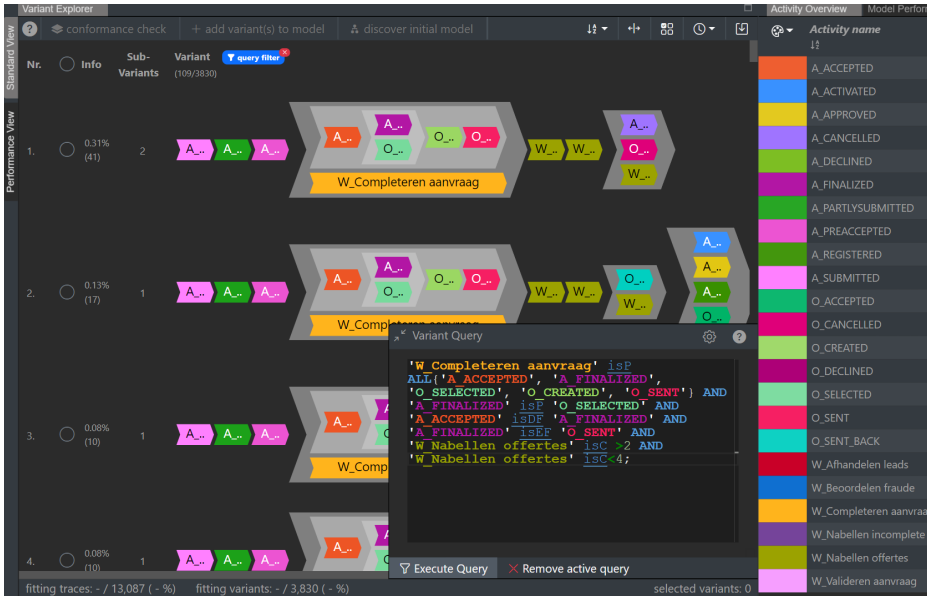


Fig. 3: Excerpt from a screenshot of Cortado showing a query editor (bottom right), a trace variant explorer visualizing the matching trace variants of the query, and a tabular overview of activities from the event log

[3] Available at https://cortado.fit.fraunhofer.de/

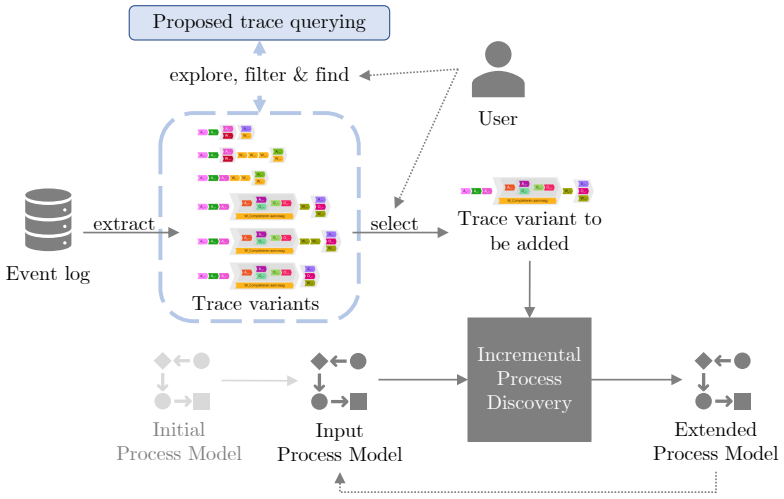


Fig. 4: Example of an application scenario of the proposed query language, i.e., trace variant selection in the context of incremental process discovery

Fig. 3 depicts a screenshot of Cortado. The shown chevron-based visualizations represent *trace variants*[4] from the loaded event log that satisfies the displayed query. We refer to [13] for an introduction to the trace variant visualization. As shown in Fig. 3, the query editor offers syntax highlighting; colors of the activity labels in the query editor correspond to the colors used in the variant explorer to improve usability. Executing a query results in an updated list of trace variants satisfying the query. In Fig. 3, the numbers at the top next to the blue filter icon indicate that 109 out of 3,830 trace variants satisfy the displayed query. In the backend, we use ANTLR [7] for generating a parser for the query language. The language's design ensures that every valid query, when parsed with ANTLR, corresponds to a single parse tree that can be transformed into a unique query tree (cf. Fig. 2).

5 Application Scenario Example

This section presents an exemplary application scenario of the proposed query language. *Process discovery* is concerned with learning a process model from an event log. Conventional discovery approaches [1] are fully automated, i.e., an event log is provided as input and the discovery algorithm returns a process model describing the event data provided. Since automated process discovery algorithms often return process models of low quality, *incremental/interactive process discovery* approaches have emerged [15] to additionally utilize domain

[4] A trace variant summarizes traces that share identical ordering relationships among the contained activities.

knowledge next to event data. Incremental process discovery allows users to gradually add selected traces to a process model that is considered under construction. By building a process model gradually, users can control the discovery phase and intervene as needed, for example, by selecting different traces or making manual changes to the model. In short, gradually selecting traces from event data is the major form of interaction in incremental process discovery, cf. Fig. 4.

With event logs containing numerous trace variants, user assistance in exploring, finding, and selecting trace variants is critical for the success of incremental process discovery. For instance, the log used in Fig. 3 contains 3,830 trace variants. Manual visual evaluation of all these variants is inappropriate. In such a scenario, the proposed query language is a valuable tool for users to cope with the variety, complexity, and amount of trace variants. As most process discovery approaches [1], including incremental ones, focus on learning the control flow of activities, a specialized query language focusing on control flow constraints is a valuable tool. To this end, we implemented the query language in Cortado, a tool for incremental process discovery, cf. Fig. 4.

6 Evaluation

This section presents an evaluation focusing on performance aspects of the query language. Sect. 6.1 presents the experimental setup and Sect. 6.2 the results.

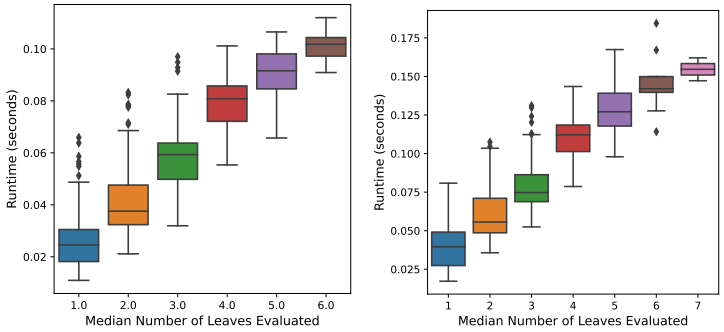
6.1 Experimental Setup

We used four publicly available, real-life event logs, cf. Table 3. For each log, we automatically generated queries from which we pre-selected 1,000 such that no finally selected query is satisfied by all or by no trace in the corresponding log. With this approach, we have attempted to filter out trivial queries to evaluate. We measured performance-related statistics given the 1,000 queries per log.

Table 3: Statistics about the event logs used

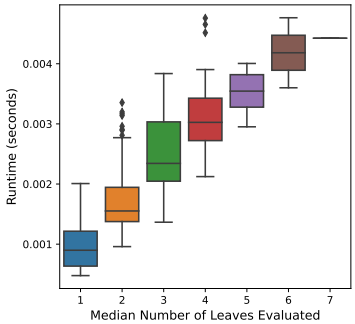
Event Log	**#Traces**	**#Trace Variants**[a]
BPI Challenge 2012[b]	13,087	3,830
BPI Challenge 2017[d]	31,509	5,937
BPI Challenge 2020, Prepaid Travel Cost log[d]	2,099	213
Road Traffic Fine Management (RTFM)[e]	150,370	350

[a] Based on the variant definition presented in [13]
[b] https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
[d] https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
[d] https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51
[e] https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

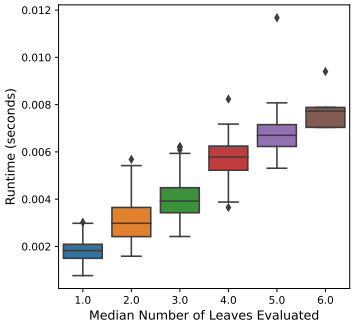


(a) BPI Challenge 2012

(b) BPI Challenge 2017



(c) BPI Challenge 2020



(d) RTFM

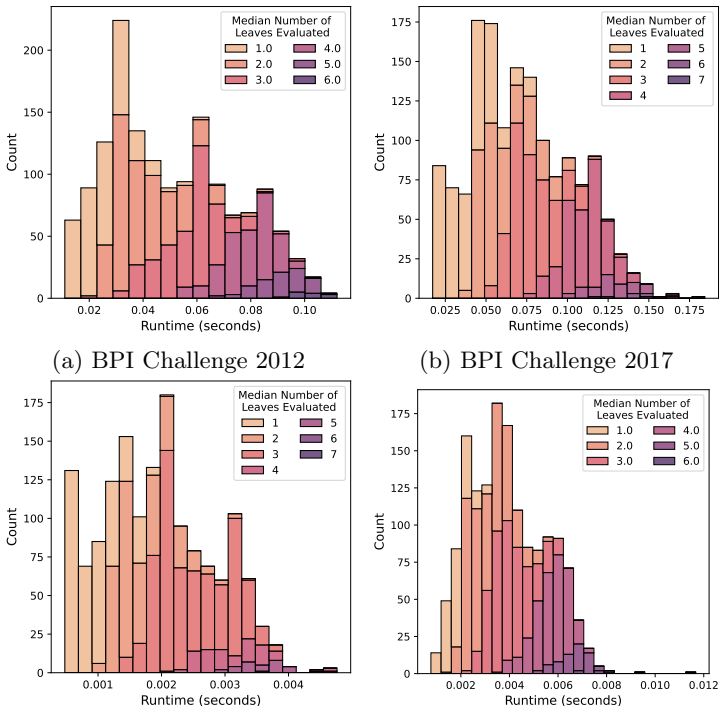
Fig. 5: Query evaluation time. Since the queries are applied to all traces, they are ordered by the median number of leaves evaluated per trace

6.2 Results

Each query is applied to all traces from the given event log. Since not all leaves of a query have to be evaluated, cf. Sect. 4.3, the number of leaves evaluated may differ per trace. Thus, the actual trace determines how many leaves of a given query must be evaluated. Fig. 5 shows the runtime (in seconds) of the queries per event log for the median number of leaf nodes that were evaluated. Thus, each boxplot is made up of 1,000 data points, i.e., 1,000 queries each evaluated on all traces from the given log. Across all four event logs, we clearly observe a linear trend of increasing runtime the more query leaves are evaluated.

Fig. 6 depicts the distribution of queries according to their evaluation time. Further, we can see the proportion of leaves evaluated at the median. As before, each plot contains 1,000 data points, i.e., 1,000 queries. Similar to Fig. 5, we observe that the number of evaluated leaves is the primary driver of increased evaluation time. The observed behavior is similar for the different logs.

Fig. 7 shows the impact of early termination, as introduced in Sect. 4.3. Note that in the previous plots, i.e., Fig. 5 and Fig. 6, early termination was always used. We clearly see from the plots in Fig. 7 that early termination has



(a) BPI Challenge 2012 (b) BPI Challenge 2017

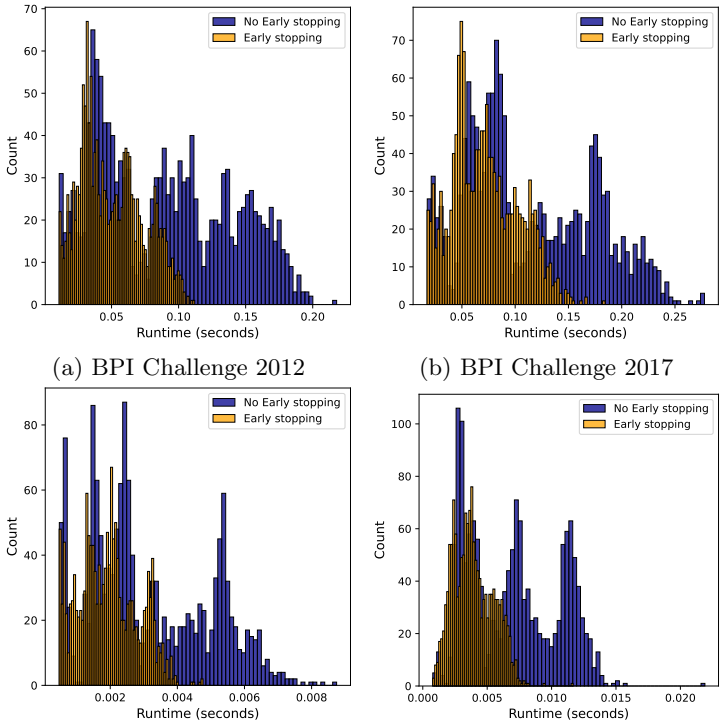
(c) BPI Challenge 2020 (d) RTFM

Fig. 6: Query evaluation time distribution

a significant impact on the evaluation time of a query across all used event logs. In conclusion, the results shown in this section indicate that the time required to evaluate queries increases linearly with the number of leaves evaluated.

7 Conclusion

We proposed a novel query language that can call up traces from event logs containing partially ordered event data. The core of the language is the control flow constraints, allowing users to specify complex ordering relationships over executed activities. We formally defined the query language's syntax and semantics. Further, we showed its implementation in the tool Cortado. We presented one potential application scenario of the language, i.e., the trace selection within incremental process discovery. In short, the proposed query language facilitates handling large event logs containing numerous traces consisting of partially ordered activities. For future work, we plan to conduct user studies exploring the query language's ease of use [12]. Further, we plan to extend the language with a graphical editor allowing query specification in a no-code environment.



(a) BPI Challenge 2012 (b) BPI Challenge 2017

(c) BPI Challenge 2020 (d) RTFM

Fig. 7: Impact of early termination on the query evaluation time

References

1. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Transactions on Knowledge and Data Engineering **31**(4), 686–705 (2019). https://doi.org/10.1109/TKDE.2018.2841877
2. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with bp-ql. Information Systems **33**(6), 477–507 (2008). https://doi.org/10.1016/j.is.2008.02.005
3. Beheshti, S.M.R., Benatallah, B., Motahari-Nezhad, H.R., Sakr, S.: A query language for analyzing business processes execution. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) Business Process Management, Lecture Notes in Computer Science, vol. 6896, pp. 281–297. Springer (2011). https://doi.org/10.1007/978-3-642-23059-2_22
4. Kobeissi, M., Assy, N., Gaaloul, W., Defude, B., Haidar, B.: An intent-based natural language interface for querying process execution data. In: 2021 3rd International Conference on Process Mining (ICPM). pp. 152–159. IEEE (2021). https://doi.org/10.1109/ICPM53251.2021.9576850
5. Markovic, I., Costa Pereira, A., de Francisco, D., Muñoz, H.: Querying in business process modeling. In: Di Nitto, E., Ripeanu, M. (eds.) Service-Oriented Computing

- ICSOC 2007 Workshops, Lecture Notes in Computer Science, vol. 4907, pp. 234–245. Springer (2009). https://doi.org/10.1007/978-3-540-93851-4_23
6. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer **40**(11), 38–45 (2007). https://doi.org/10.1109/MC.2007.400
7. Parr, T.J., Quong, R.W.: Antlr: A predicated-ll(k) parser generator. Software: Practice and Experience **25**(7), 789–810 (1995). https://doi.org/10.1002/spe.4380250705
8. Polyvyanyy, A.: Business process querying. In: Sakr, S., Zomaya, A.Y. (eds.) Encyclopedia of Big Data Technologies, pp. 1–9. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-77525-8_108
9. Polyvyanyy, A.: Process Querying Methods. Springer (2022). https://doi.org/10.1007/978-3-030-92875-9
10. Polyvyanyy, A., Ouyang, C., Barros, A., van der Aalst, W.M.P.: Process querying: Enabling business intelligence through query-based process analytics. Decision Support Systems **100**, 41–56 (2017). https://doi.org/10.1016/j.dss.2017.04.011
11. Räim, M., Di Ciccio, C., Maggi, F.M., Mecella, M., Mendling, J.: Log-based understanding of business processes through temporal logic query checking. In: Meersman, R., Panetto, H., Dillon, T., Missikoff, M., Liu, L., Pastor, O., Cuzzocrea, A., Sellis, T. (eds.) On the Move to Meaningful Internet Systems: OTM 2014 Conferences, Lecture Notes in Computer Science, vol. 8841, pp. 75–92. Springer (2014). https://doi.org/10.1007/978-3-662-45563-0_5
12. Reisner, P.: Human factors studies of database query languages: A survey and assessment. ACM Computing Surveys **13**(1), 13–31 (1981). https://doi.org/10.1145/356835.356837
13. Schuster, D., Schade, L., van Zelst, S.J., van der Aalst, W.M.P.: Visualizing trace variants from partially ordered event data. In: Munoz-Gama, J., Lu, X. (eds.) Process Mining Workshops, Lecture Notes in Business Information Processing, vol. 433, pp. 34–46. Springer (2022). https://doi.org/10.1007/978-3-030-98581-3_3
14. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Cortado—an interactive tool for data-driven process discovery and modeling. In: Buchs, D., Carmona, J. (eds.) Application and Theory of Petri Nets and Concurrency, Lecture Notes in Computer Science, vol. 12734, pp. 465–475. Springer (2021). https://doi.org/10.1007/978-3-030-76983-3_23
15. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Utilizing domain knowledge in data-driven process discovery: A literature review. Computers in Industry **137** (2022). https://doi.org/10.1016/j.compind.2022.103612
16. van der Aalst, W.M.P.: Service mining: Using process mining to discover, check, and improve service behavior. IEEE Transactions on Services Computing **6**(4), 525–535 (2013). https://doi.org/10.1109/TSC.2012.25
17. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer (2016). https://doi.org/10.1007/978-3-662-49851-4
18. Vogelgesang, T., Ambrosy, J., Becher, D., Seilbeck, R., Geyer-Klingeberg, J., Klenk, M.: Celonis pql: A query language for process mining. In: Polyvyanyy, A. (ed.) Process Querying Methods, pp. 377–408. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-92875-9_13
19. Wang, J., Jin, T., Wong, R.K., Wen, L.: Querying business process model repositories. World Wide Web **17**(3), 427–454 (2014). https://doi.org/10.1007/s11280-013-0210-z

20. Yongsiriwit, K., Chan, N.N., Gaaloul, W.: Log-based process fragment querying to support process design. In: 2015 48th Hawaii International Conference on System Sciences. pp. 4109–4119. IEEE (2015). https://doi.org/10.1109/HICSS.2015.493