Journal on Data Semantics (2021) 10:109–141
https://doi.org/10.1007/s13740-021-00122-1

ORIGINAL ARTICLE

Multi-Dimensional Event Data in Graph Databases

Stefan Esser[1] · **Dirk Fahland**[2]

Received: 29 May 2020 / Revised: 12 November 2020 / Accepted: 6 March 2021 / Published online: 27 May 2021

Abstract
Process event data is usually stored either in a sequential process event log or in a relational database. While the sequential, single-dimensional nature of event logs aids querying for (sub)sequences of events based on *temporal relations* such as "directly/eventually-follows," it does not support querying *multi-dimensional* event data of multiple related entities. Relational databases allow storing multi-dimensional event data, but existing query languages do not support querying for sequences or paths of events in terms of temporal relations. In this paper, we propose a general data model for multi-dimensional event data based on *labeled property graphs* that allows storing structural and temporal relations in a single, integrated graph-based data structure in a systematic way. We provide semantics for all concepts of our data model, and generic queries for modeling event data over multiple entities that *interact synchronously and asynchronously*. The queries allow for efficiently converting large real-life event data sets into our data model, and we provide 5 converted data sets for further research. We show that typical and advanced queries for retrieving and aggregating such multi-dimensional event data can be formulated and executed efficiently in the existing query language Cypher, giving rise to several new research questions. Specifically, aggregation queries on our data model enable process mining over multiple inter-related entities using off-the-shelf technology.

Keywords Process mining · Event log · Multi-dimensional processes · Querying · Labeled property graphs · Graph databases

1 Introduction

Retrieving and aggregating subsets of event data of a particular characteristic is a recurring activity in process analysis and process mining [1]. Each *event* is thereby defined by an *event classifier* such as the *activity* or state that was recorded, a *case identifier* referring to the object or entity where the activity was carried out, and a *timestamp* or *ordering* attribute defining the order of events.

If all events use the same single case identifier attribute, then the event data has a *single behavioral dimension*. It can be stored in an *event log* as one *sequence of events* per case according to the data model of the XES-Standard [2]. Such sequences can be easily queried for *behavioral properties* such as *event (sub-)sequences* or *temporal relations* such as "directly/eventually-follows" in combination with other data attributes [3–8]. *Aggregating* directly/eventually-follows relations between events is fundamental for discovering process models from event logs [1,9,10].

Most processes in practice, however, involve multiple inter-related entities. In such data, each event is directly or indirectly linked to multiple different case identifiers and is part of multiple behaviors or dynamics [11]. The data has *multiple behavioral dimensions*. Sequential event logs cannot represent these multiple behavioral dimensions correctly [12]. Relational databases (RDBs) can store *1:n and n:m relations between events and case identifiers* and among different case identifiers. However, tables in an RDB cannot represent sequences of events of arbitrary length explicitly. The explicit behavioral information is lost and hence cannot be queried in a natural way [13,14].

State of the art Although sequential event logs and RDBs enjoy standardized data models built on a few basic concepts, they cannot easily be combined for correct information representation *and* intuitive querying. Extracting event data from RDBs into sequential event logs requires large non-intuitive queries [11,14] and introduces false information [12]. Direct behavioral queries on RDBs are limited to a single behav-

✉ Dirk Fahland
d.fahland@tue.nl

Stefan Esser
stefan.esser@inform-software.de

[1] INFORM GmbH, Aachen, Germany

[2] Eindhoven University of Technology, Eindhoven, The Netherlands

ioral dimension [15,16]. Extensions that integrate relational data into the event log or relate case identifiers of multiple sequential event logs to each other [12,14,17–19] cannot be queried across multiple behavioral dimensions due to the strict sequential nature of event logs.

Graph-based data models for event data [20–23] have been proposed as alternative to overcome these limitations. Graphs can describe relations between various entities *and* sequential information as paths in one data structure. While all existing graph-based data models share similarities, no work systematically identified a *minimal* set of core concepts and the necessary semantics needed to model, query, and aggregate event data over multiple behavioral dimensions; see Sect. 2 for details.

Research problem In this paper, we approach the problem of identifying a generally applicable model of event data in a multi-dimensional setting. The specific problem is to *identify a minimal set of core concepts for a data model of multi-dimensional process event data with clearly defined semantics* to fully (1) model, (2) query, and (3) aggregate all kinds of real-life process event data suitable for process analysis. That data model and queries have to support known requirements for analyzing multi-dimensional event data [12,13,24] and be realizable in existing off-the-shelf *graph database systems* [25].

Method First, we determined the process event data concepts and requirements any data model had to support based on literature (see Sects. 2.1 and 2.2). We then analyzed existing data models regarding these requirements (see Sect. 2.3). All recent works that succeed in modeling (some) aspects of multi-dimensional event data employ graph properties. We therefore took the most complete proposal [22] based on *labeled property graphs* (LPGs, see Sect. 2.4) as a starting point.

To ensure our model would support real-life process event data, we identified from a collection of public real-life event logs[1] 5 data-sets with unique multi-dimensional characteristics that can serve as benchmark: multiple entities interact via shared common entity (BPIC14 [26]); multiple entities interact asynchronously, based on click-stream data (BPIC16 [27]), based on a case management system (BPIC17 [28][2]), based on ERP system data (BPIC19 [30]); multiple event logs of the same processes executed in different organizations, BPIC15 [31]. A data model has to allow modeling, querying, and aggregating at least these 5 data sets in their multi-dimensional nature.

We then iteratively developed a data model and queries that could support all 5 benchmark data sets on the existing graph database system Neo4J (neo4j.com); Neo4j was chosen for LPG storage and querying due to off-the-shelf availability and suitable performance. We used an *Extract-Load-Transform* [32,33] approach:

1. We *extracted* the event data into a standard event table where each record describes one event and its properties, including references to all entities involved. All event data can be extracted into this format; see Sect. 2.3.
2. We *loaded* all events into the GDB as an LPG of unrelated raw event nodes.
3. We identified the domain concepts stored in each data set, specifically entities and relations between entities and events stored in the event attributes.
4. We iteratively identified semantic node types and relationship types for LPGs to abstract the domain concepts identified in step 3. Our semantic concepts for nodes and relations thereby had to serve as adequate abstractions so that all event data could be queried and analyzed through these semantic abstractions only.
5. We then developed queries to *transform* the raw events of step 1 into a graph that uses the identified LPG concepts of step 4.
6. In case a suitable solution (node types, relationship types, queries) was found for one data set, we applied it on all other data sets. If the solution could not be applied on one data set, we identified the cause and generalized the concepts and queries and repeated steps 3–5 for all data sets.

We conducted over 100 iterations of the above process over all 5 data sets until reaching a fixed point.

Contribution and results We contribute a generally applicable, minimal, integrated data model for *event data in multiple behavioral dimensions* in labeled property graphs. We identified

1. *4 semantic node types* for multi-dimensional event data in LPGs: (i) events, (ii) entities, (iii) logs, and (iv) event classes;
2. *3 semantic structural relations* for relating each event (i) to one or more entities, (ii) to exactly one log, and (iii) to one or more event classes;
3. And *2 semantic behavioral relations* describing (i) directly-follows between two events (along a chosen entity), and its congruent (ii) directly-follows relation between event classes (summarizing event-level directly-follows on class level)

[1] https://data.4tu.nl/repository/collection:event_logs_real.

[2] BPIC18 [29] originates from a similar kind of system and has the same characteristics as BPIC17 and hence was excluded.

See Sect. 3 for the concepts and Sect. 4 for their semantic definition in terms of LPGs. Our model allows querying and aggregating the modeled multi-dimensional event data and thereby subsumes and exceeds several prior works.

4. We identified queries to extract entities and relations between entities from raw events, and to reify relations into composite entities,
5. To correlate events to entities,
6. To derive directly-follows relations for all events of an entity or relation.

All the available multi-dimensional event data could be transformed into our model using a standard set of queries; see Sect. 5. The resulting graphs are available for further research [34–38].

7. We show that existing query languages for labeled property graphs using our event data model satisfy all requirements for querying event data over multiple behavioral dimensions known from the literature [13,24].

We evaluated the query results to be correct against a manually constructed ground truth. Query execution times are on par or faster compared to commercially available event data pre-processing techniques on single-dimensional event data. Query execution times perform hand-written algorithms on multi-dimensional event data. We evaluated the expressive power of the query language used against existing process query languages [39,40] and identify potential for further research; see Sect. 6. We identified

8. Queries to aggregate event-level directly-follows relations into a directly-follows graph per entity, including filtering, thereby enabling basic process discovery;
9. Queries to derive and aggregate directly-follows relations between related entities into a directly-follows graph describing entity interaction (through a reified entity).
10. The directly-follows graphs of different entities and relations share nodes. We thereby obtain a method for discovering process models which describe the behavior of a network of related entities and their interactions as a single *multi-viewpoint model* [21].

In Sect. 7, we demonstrate discovery of artifact-centric models of multiple entities with asynchronous and synchronous interactions [12], also called multi-viewpoint models.

All queries realizing the transformations into our data model, querying the data, and discovering process models through graph databases are available at GitHub[3] and at [41]. We discuss limitations and alleys for future work in Sect. 8.

2 Background

Information systems (ISs) create and update information records in structured transactions or *activities* throughout process executions. Each update can be recorded as an *event*. Event attributes describe the activity carried out and the *timestamp* (or *ordering* of updates). Furthermore, each event is linked to one or more *entities* on which the updates occurred via unique *entity identifier* attributes, for example, a credit application order and related credit offers. The entities themselves are related to each other via structural 1:1, 1:n, and n:m relations [1,12].

We first recall the foundational concepts of single-dimensional process event logs in Sect. 2.1. After discussing challenges and requirements for analyzing multi-dimensional event data in Sect. 2.2, we discuss the state of the art on modeling, querying, and analyzing multi-dimensional event data in Sect. 2.3, before we recall the data model of labeled property graphs and the query language Cypher in Sect. 2.4.

2.1 Modeling Single-Dimensional Event Logs

A *process event log* is a collection of recorded events E structured into a specific *view* on an information system from the of perspective handling of *one* specific entity, e.g., handling a credit application. Table 1 shows a simplified event log taken from the BPIC17 [28] data set describing the handling of a credit application (identified by attribute *Appl.*).

The following process-specific concepts are part of process event logs [1].

E1 **Event, activity, and timestamp** Each event $e \in E$ in an event log records an *atomic* observation of an activity name $e.activity$ that has been observed at a particular point in time $e.time$ (or an *ordering attribute that orders the events in the log*), e.g., *Activity* and *Timestamp* in Table 1.

E2 **Entity identifier, correlation, case** Each event records at least one identifier $e.entityid$ of some entity, e.g., a specific credit application (*Appl.*) or credit offer (*oID*) in Table 1. The set of events *correlated* to the same entity identified by id is $\{e_1, \ldots, e_n\} = \{e \in E \mid e.entityid = id\}$. All events correlated to the same entity belong to the same *case* (or execution) of the process.

[3] https://github.com/multi-dimensional-process-mining/graphdb-eventlogs.

Table 1 Simplified BPIC'17 Example

	Appl.	Activity	Timestamp	Res.	Amount	oID	Origin
e_1	1	**C**reate **A**ppl.	29.08.19 10:30	10	1000		A
e_2	1	**A**ppl. **R**eady	29.08.19 10:35	10	1000		A
e_3	1	**H**andle **L**eads	29.08.19 10:40	42	1000		W
e_4	1	**C**reate **O**ffer	29.08.19 13:14	11	1000	1	O
e_5	1	**S**end **O**ffer	29.08.19 13:35	11	1000	1	O
e_6	1	**C**reate **O**ffer	30.08.19 08:49	12	1000	2	O
e_7	1	Offer **Ca**nceled	30.08.19 09:05	10	1000	1	O
e_8	1	**S**end **O**ffer	30.08.19 09:20	12	1000	2	O
e_9	1	**C**ontact **C**ust.	30.08.19 10:15	44	1000		W
e_{10}	1	Offer **Ret**urned	30.08.19 15:20	16	1000	2	O

E3 **Life-cycle information** An optional attribute $e.lifecycle$ records states of long-running behavior, e.g., an activity *started* or *completed*.

E4 **Resource** An optional attribute $e.resource$ records whether an actor or resource was involved in the event, e.g., *Res.* in Table 1.

E5 **Trace** The sequence $\langle e_1, \ldots, e_n \rangle$ of all events correlated with an entity ordered by time (or the ordering attribute) is called a *trace* (of this entity), e.g., $\langle e_1, \ldots, e_{10} \rangle$ is the trace of "Application 1" in Table 1.

E6 **Event class** Besides the activity, also other attributes or combinations of attributes can be designated as "representative" for an event. The notion of *event class* generalizes the notion of the activity attribute (E1); the analyst can designated one (or more) event attributes as event class $e.class$ depending on the analysis, e.g., we can set $e.class := e.Res$ in Table 1 as event class when analyzing handover-of-work behavior between users.

The IEEE XES-Standard [2] materializes these concepts in a tree-structure that specifically pre-determines a unique case identifier and available event classes.

The behavior recorded in a sequential event log can be analyzed by querying and aggregating the *directly-follows* relation over events E [1].

E7 **Directly-follows (events)** Event e_b *directly-follows* e_a, $e_a \rightarrow e_b$ iff there is a trace $\langle \ldots, e_a, e_b, \ldots \rangle$, e.g., e_7 directly follows e_6 in Table 1.

E8 **Directly-follows (event classes)** We can aggregate the directly-follows relation over event classes: event class b directly-follows event class a, $a \rightarrow b$ iff there is a trace $\langle \ldots, e_a, e_b, \ldots \rangle$ with $e_a.class = a$ and $e_b.class = b$. For example, *Offer Canceled* directly-follows *Create Offer* for $e.class = e.Activity$ and 10 directly-follows 12 for $e.class = e.Res$.

Nearly, all process discovery algorithms create for each event class one activity node [1]. Then, dependencies between activity nodes in the model can be derived from the directly-follows relation between activities in the event log [9,10].

2.2 Requirements for Analyzing Multi-Dimensional Event Data

An information system usually hosts multiple uniquely identifiable entities [12], e.g., credit applications and offers. For example, the BPIC'17 data (shown in Table 1) identifies four entities: credit applications (identified by *Appl.*, events with $Origin = A$), credit offers (identified by *oID* with $Origin = O$) with a 1:n relation to Applications, the Workflow (identified by *Appl.* with $Origin = W$); and the actors working on the case (identified by *Res.*) with an n:m relation to Application, Workflow, and Offers.

Relational databases allow recording events, their correlations to entities, and the 1:n and n:m relations between entities as partly illustrated in Fig. 1(3) and are therefore the default storage for event data in information systems. Analysis of the behavior, however, requires extraction of the data into a sequential format [11]. Extracting a single-dimensional event log [Fig. 1(1)] correlates all events under a single entity (case identifier), e.g., the Application or the Offer document, and *flattens* the data accordingly [11] leading to false behavioral information called *convergence* and *divergence* [12,17].

Convergence Flattening the data in Fig. 1(3) under *Application* de-normalizes the 1:n relation to *Offer* and results in the event log of Table 1 and Fig. 1(1): We observe *Cancelled* (e_7) directly followed by *Send Offer* (e_8), suggesting that an offer was sent after it has been cancelled. However, this sequence of activities never happened for any entity in the data: e_7 refers to $oID = 1$, whereas e_8 refers to $oID = 2$. The entire BPIC17 event log contains 15% such false directly-follows pairs of events while missing over 50% of the actual directly-follows pairs. Some events are only involved in false directly-follows pairs; see Appendix A. The false directly-

follows pairs cannot be removed from sequential event logs [17] and can amount to over 50% of all pairs rendering the analysis useless [12].

Divergence Flattening the data in Fig. 1(3) under *Offer* (*oID*) via the n:1 relation replicates events on the 1-side for each entity on the n-side, see Fig. 1(1). The two traces for $o1$ and $o2$ contain the events e_1, e_2, e_3, e_9 which are only indirectly correlated to the offer (via their parent application). The resulting event log contains multiple copies of the same event and more directly-follows pairs than exist in reality [17].

To avoid both phenomena, we formulate requirement **(R0)**: Any event data model for event data over multiple related entities has to describe directly-follows relations between pairs of events (e_1, e_2) only along the entities to which e_1 and e_2 are both correlated.

A recent literature survey of 95 studies [13,24] established further requirements for *modeling and querying* event data. From this survey, we identified the following requirements that specifically address *querying for structure and behavior in multi-dimensional event data* [24, pp.133]: **(R1)** to query and analyze events (E1 of Sect. 2.1), and **(R2)** to consider relations between multiple data entities (as in RDBs). The technique shall support **(R3)** storing and querying business process-oriented concepts (E3-E4) and **(R4)** capture information about how events are correlated with different entities (E2) to avoid convergence and divergence (generalize E5-E8 so that R0 is satisfied).

According to [13,24], queries should **(R5)** be expressed as graphs to specify the behavior of interest in a natural way, **(R6)** allow to query paths (or sequences) of events (connected by some relation), **(R7)** allow to select individual cases based on partial patterns, **(R8)** allow to query temporal properties (such as directly/eventually-follows), **(R9)** correlate events related to the same entity, **(R10)** allow querying aspects related to several entities or processes at the same time on the same data set, and **(R11)** allow to query multiple event logs and combine results.

Prior work on analyzing multi-dimensional event data [12,42] identified four major aggregation operations for discovering so-called *artifact-centric process models*. The technique has to support **(R12)** aggregating events into *user-defined activities*, viz. event classes (see E6 in Sect. 2.1) based on event properties (e.g., to distinguish different types of "update" activities based on the nature of the update). Moreover, it has to support **(R13)** materializing a relation between two events as a new derived entity, as this allows to model, query, and aggregate *interactions* between different entities. Further, it has to allow **(R14)** aggregating directly-follows relations from the event level to the *event class* level *per entity type*, and **(R15)** relating or synchronizing aggregated behavior of different entity types.

Altogether, *a user shall be able to query and aggregate for individual events (and their properties), for different entities/case notions, for behavioral and structural relations, and for patterns of multiple events (within and across entities).*

2.3 Related work

We review 5 existing types of data models for event data against the requirements of Sect. 2.2 showing that *no existing data model or query language on sequential event logs or RDBs satisfies R0–R15*. Moreover, we discuss related work on behavioral process query languages.

2.3.1 Single event log for a single, selected entity

Event logs as described in Sect. 2.1 and illustrated in Fig. 1(1) cannot correctly model or aggregate behavior over events related to multiple entities due to convergence and divergence as discussed in Sect. 2.2, thus violating R0. Sequential event logs can be stored and queried using files [2] or through RDBs [43]. XES event logs fall into this type of data models [2].

Of the 95 works surveyed [24, pp.133], several approaches exist to retrieve cases from event logs for temporal properties [5,7], for most frequent behavior [4], for sequences of activities [3] or algebraic expressions of sequence, choice, and parallelism over activities [8], or to check whether a temporal-logic property holds [6]. Several techniques support graph-based queries [5,44,45]. These techniques satisfy R7 and R8. However, they only support a single fixed case notion and thus fail R2, R10, R11.

2.3.2 Event table with multiple entity identifiers.

The model is a variant of Table 1 and Fig. 1(2) and defines a single table. Each record is an event with *multi-valued* entity identifier attributes. The model was first introduced by Popova et al. [19] and later formalized by Aalst et al. as *object-centric log* [17]. *Redo* event logs [46] include database operations. *XOC* logs [18] even include database snapshots. The BPAF [47] format is a precursor that allows querying event data of different processes [48] but not based on properties of specific events (violates R7). All these models only describe correlation (and data operations) of an event to multiple entities, but leave sequential ordering per entity implicit, see Fig. 1(2), which violates R0 and prevents R6 and R8. They are usually transformed to other formats for analysis [19,21].

2.3.3 Event data in a relational database.

Events are stored as time-stamped attributes and can be related to various entities through primary and foreign keys as shown in Fig. 1(3). Dijkman et al. [15] show a native, effi-

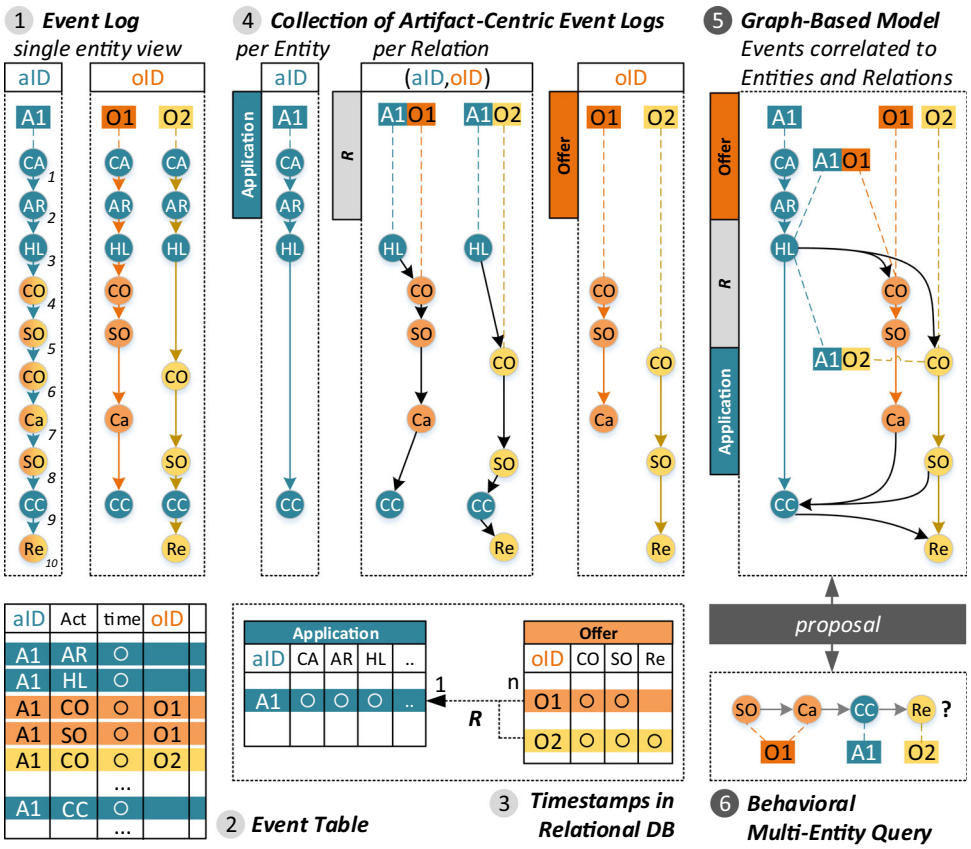


Fig. 1 Illustration of event data models on the example events of Table 1

cient relational algebra operator to query directly-following events. Also, pre-defined behavioral patterns can be queried efficiently [16]. However, these operators are fixed to one entity identifier and querying paths requires unbounded joins (violates R4,R6,R10).

2.3.4 Multiple logs, one per entity and per relation

Convergence and divergence can be avoided by extracting one log per entity providing multiple views [12,14,17,19], thus satisfying R0. For example, Fig. 1(4) shows a log for the Application and a log for both Offers. Sets of logs with disjoint sets of events can be extracted automatically by partitioning the relational schema [12,19].

Interactions between entities can be modeled by extracting a sequential log per relation: per record in the relation, order all events correlated to the entities in the relation by time, the obtained trace describes the entities' interaction [12]. For example, in Fig. 1(4), the log for the relation R contains traces for the interaction $(A1, O1)$ between the Application and Offer 1 and for the interaction $(A1, O2)$ between the Application and Offer 2. Extraction of an interaction log corresponds to reifying the relation into an entity which overlaps and synchronizes with other entities [42]. The approach of [14] provides a meta-model to extract event logs of different perspectives from user-defined, composite entities [14] that also may overlap. However, the separation into multiple event logs violates R9 and R10, and if logs do not overlap also R15.

DAPOQ [24, Ch.7] generalizes various prior query languages to query and extract events in the context of their relational data model for behavior properties, but does not support retrieving individual cases (R7) or specifying behavioral and structural patterns (R8).

2.3.5 Graph-based, events as nodes related to multiple entities

One of the first graph-based data models for event data is built on RDF [20,49,50]. The data model expresses events (as nodes) and relations between events and entities, and between different entities (as edges). This graph can be queried using SPARQL, satisfying R1–R5 and R9,R10. A scalable service-oriented architecture allows to retrieve event logs through a regular-expression describing the behavior of interest wrt. various entities [20] which satisfies R7 and R8. OLAP operations allow exploring multiple data dimensions [49]. The technique, however, always materializes the result as paths wrt. a single entity identifier (violates R0). Our proposed data model bears many similarities to this data model.

The main difference is that we explicitly store events and directly-follows pairs between events wrt. the entities they are correlated to. We thereby obtain an explicit partial-order event data model that treats events and entities as first-class citizens as shown in Fig. 1(5). This allow us to model and query multiple *behavioral dimensions* at once, for example, a path *Send Offer* → *Offer Cancelled* → *Contact Customer* → *Offer Returned* over 3 different entities as shown in Fig. 1(6). In contrast, the classical notion of multi-dimensionality in event data refers to attributes of entities and events [49]. Finally, we use labeled property graphs (LPGs) instead of RDF, which allows to reduce the graph size as LPGs allow storing key/value pairs within a node or relationship.

Werner et al. [23] modeled behavior over two entity types in financial auditing as a *graph over events* describing "directly-follows" per entity or relation (satisfying R0–R4), but their model was not generalized or used for querying (violates R5–R11). Our graph-based model [22] shown in Fig. 1c generalizes the model of Werner et al. [23] to standard process concepts (Sect. 2.1) using labeled property graphs and Cypher [51].

Berti et al. [21] convert object-centric logs (Sect. 2.3.2) into two separate graphs: One describes correlation of events to entities, and one describes the directly-follows relation between any two events per entity similar to [22,23]. Assuming that all relations between entities have been reified into entities (see Sect. 2.3.4), they aggregate the directly-follows relations per entity and satisfy R14, R15. However, in most event data in practice [12,28], such as Fig. 1, relations are not reified yet, limiting the applicability of [21] as it does not support R13. Further, the model does not support querying the event data prior to aggregation (violates R6, R7, R8, R10) because correlation and directly-follows relations are stored in separate graphs.

In a prior exploratory case study [22], we presented an integrated data model for BPIC17 [28] based on *labeled property graphs*. We used edges to correlate events to entities and to model "directly-following" events per entity as shown in Fig. 1(5). We used *graph query languages* of existing *graph database systems* [25] to answer behavioral multi-entity queries [Fig. 1(6)] and for aggregating directly-follows relations for social network analysis, satisfying R0–R11 and R13. However, our data model did not provide a generic data model for various real-life data sets and did not support R11, R12, and R14.

In this paper, we generalize our previously explored integrated, graph-based model [22] shown in Fig. 1(5) to be applicable to all kinds of real-life data sets while satisfying R0–R11 and additionally R12–R15, thereby subsuming prior works.

Behavioral process query languages. The problem of querying behavior has also been studied for the use case of

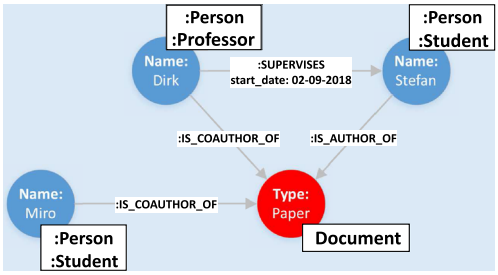


Fig. 2 Labeled property graph

retrieving a process model M for a model collection $\mathcal{M}$. The query language PQL allows specifying behavior in the form of example patterns [39] or behavioral relations [40]. The query returns all models M whose behavior intersects with the specified behavior. In contrast, querying event logs means to retrieve *all* sub-graphs over events, i.e., all concrete process executions, that match the specified behavior. Moreover, in our use case the behavior is explicitly stored, whereas in process model querying the behavior first has to be (partially) computed from the process model syntax. Although both use cases are significantly different, the behavioral query constructs developed in PQL are of general nature and could be applied to this setting. However, PQL is currently not designed to be evaluated against an event data model, whereas Cypher and Neo4J are available off-the-shelf techniques. We discuss how our query language compares to PQL conceptually in Sect. 6.

2.4 Labeled property graphs and querying

We recall basic concepts and notation for labeled property graphs and for the query language Cypher, which we use in subsequent sections.

A *labeled property graph* (LPG) is a data structure used in graph databases (GDBs) [25]. Let K be a set of *keys*, V be a set of *values* and *Label* be a set of labels, $Label \cap V = \emptyset$. An LPG $G = (N, R, label, prop)$ consists of *nodes* N (vertices) and *relationships* R (edges) where each relationship $r \in R$ defines a directed edge $\overrightarrow{r} = (n_1, n_2) \in N \times N$ between two nodes. The labeling function $label : N \cup R \to 2^{Label}$ assigns to each node and each relationship a non-empty set of labels designating their type, e.g., $label(n) = \{Person, Student\}$. Function $prop : (N \cup R) \times K \to V$ assigns each node or relationship an arbitrary number of key-value pairs, called *properties*. For the value $prop(x, k) = v$ of a property key k of a node or relationship $x \in N \cup R$, we also write $x.k = v$, and $x.k = \perp$ if k is undefined for x.

The example LPG in Fig. 2 models the relationships between a professor and two students. The example contains nodes with the *labels :Person*, *:Professor*, *:Student* and *:Document*. The document you are currently reading is authored

by Stefan, a student supervised by Dirk who co-authors this document and say Miro is another student contributing to this paper. The "Name" of each person is a property of the *:Person* nodes; "Type" is a property of *:Document* nodes. The described relationships between the nodes can also hold properties like the starting date of a supervision. Neo4j supports multiple labels for nodes, while relationships have exactly one label.

Cypher is a language for querying LPGs [51] and supported by Neo4j. Cypher queries use pattern matching to select sub-graphs of interest. The Cypher pattern $(n:Label\ \{Prop\ :\ Value\})$ matches any node labeled :*Label* that has property *Prop* set to *Value*. Pattern $(n)-[r{:}Label]->(m)$ matches any relationship labeled :*Label* from a node n to a node m. Any combination of nodes and relations (n, r, m) that match the pattern is included in the result set; if any variable n, r, m is already bound, then only combinations including the bound nodes/relationships will be returned. We explain the Cypher query concepts used in this paper by a single (albeit inefficient) example query. For the graph in Fig. 2, we query for the longest path between "Dirk" and a student, other than "Stefan," who also works on a document that "Dirk" co-authors.

```
1 MATCH path = (s:Student)-[*]-(p:Professor {Name: "Dirk"})
2 WHERE NOT s.Name = "Stefan"
3 WITH s AS student, p AS professor, path AS paths
4 MATCH (d:Document)<-[:IS_COAUTHOR_OF]-(professor:Professor)
5 WHERE (student:Student)---(d)
6 RETURN student, d, paths, length(paths) AS pLength
7 ORDER BY pLength DESC
8 LIMIT 1
```

The *MATCH* clause retrieves pairs of nodes s and p and a $path$ between s and p that match the pattern in line 1: a Student s related to Professor p by a path $-[*]-$ of arbitrary relationships, direction, and length. The *WHERE* clause in line 2 restricts the pattern such that the student's name cannot be "Stefan." By defining the professors' name property to be "Dirk" in line 1, we also restrict the pattern. *WITH* in line 3 formats and renames the result set (e.g., s renamed to *student*) that is passed to the next query from line 4 on: variable *student* in lines 4–6 may only take values retrieved for variable s in lines 1–2, e.g., "Miro" but not "Stefan." Line 4 matches the documents Dirk coauthors, and line 5 restricts the results to documents that have a direct relationship to a student.

The *RETURN* statement in line 6 formats the result set of lines 4–5 as output. For the example graph, the $student$ is "Miro" and the document d is the "paper"; variable *paths* contains the 2 possible paths between Miro and Dirk. One walks over Stefan, and one does not. "Length()" is a function of Cypher that returns the hops needed to walk a path. Lines 7–8 sort the results by path lengths (*ORDER BY* clause) in descending order (*DESC*) and return only the first path of this ordered list (*LIMIT* 1).

Instead of *RETURN*, a Cypher query may also end with a statement $CREATE(student)<-[r{:}FOUND]-(d)$ to add a new relationship of label *:Found* from node d to node $student$; statement *MERGE* only creates the specified node/relationship if it does not exist yet; see [52] for more details.

3 Representing Multi-Dimensional Event Data in Labeled Property Graphs

Labeled property graphs introduced in Sect. 2.4 allow versatile data modeling of various concepts and relations between concepts. In this section, we propose how to model the central concepts and relations of process event data of Sect. 2.1 in labeled property graphs. Figure 3 summarizes our proposal which we explain in detail below. In Sect. 4, we *constrain* the way how the concepts and relations may occur in a labeled property graph describing event data, thereby defining the *semantics* for process concepts in terms of LPGs. In that section, we also discuss how to *refine* the proposed node and relationship types to aid in the analysis.

3.1 Modeling Events Related to Multiple Entities or Logs

We introduce node and relation types for the concepts of event, entity, and event log; together they describe the *instance-level* concepts of Fig. 3, i.e., concrete entities or recorded events.

Event We represent the core element of each event log, the *event*, as a node with label :*Event* as shown in Fig. 3. Of the three mandatory event attributes (cf. Sect. 2.1), we only model *activity* and *timestamp* as properties to event nodes, having datatypes *STRING* and *DATETIME*, respectively. We describe correlation to multiple entity identifiers through the graph structure as we explain next. The graph in Fig. 4 models 5 events of Table 1.

Entity Single-dimensional event logs fix a single entity identifier (called case identifier, cf. Sect. 2.1) to which each event is correlated. Our model abandons the notion of a case identifier in favor of the more general *Entity* concept. We model each entity as a node with the label :*Entity* as shown in Fig. 3. Its property *EntityType* describes the type of the entity. Property *ID* is the entity identifier. We require the combination of *EntityType* and *ID*, stored as property *uID*, to be unique in the entire graph similar to a primary key value in a relational database (indicated by *uID* being underlined in Fig. 3). We limit our model to plainly describing entity nodes and their relations and refer to existing work [25] for modeling their semantic nature further.

The graph in Fig. 4 models 3 entities of 3 different entity types. Each *:Entity* node represents a concrete entity related to the process, such as *Application* 1 and *Offer* 1 of our running example of Table 1.

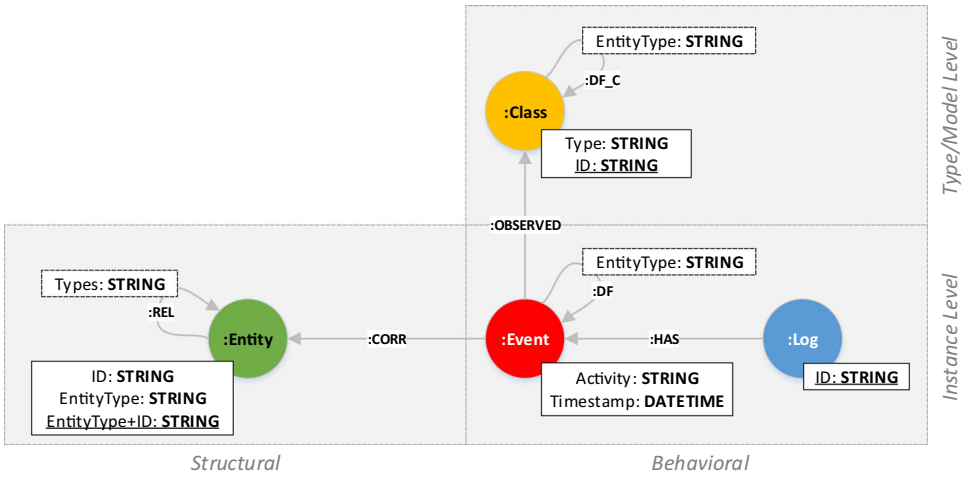


Fig. 3 Schema of node and relationship types for modeling multi-dimensional event data in labeled property graphs

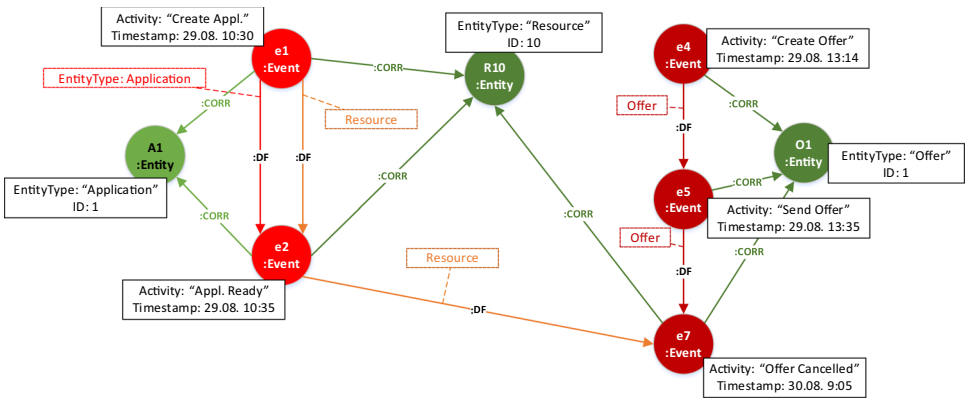


Fig. 4 Graph describing the events of Application 1, Offer 1, and Resource 10 of Table 1

Correlation We model *correlation* of an event to an entity by the dedicated relationship label *:CORR*. Through *:CORR*-relationships, we can correlate any event to any number of entities of different types, allowing for multi-dimensional correlation of events to entities as shown in Fig. 4. For example, event $e1$ is correlated to *Application 1* and *Resource 10*, whereas event $e7$ is correlated to *Offer 1* and *Resource 10*.

Event log Some event data sets, such as BPIC'15[31], consist of multiple event logs. To support multiple logs in one graph event log instance, we introduce a separate node type for logs with the label *:Log* as shown in Fig. 3. Similar to the entities, we model which event belongs to which log with the *:HAS* relationship from *:Log* to *:Event* nodes.

Attributes Figure 3 shows all properties our graph data model expects to be present. Additionally, any event, entity, or log node can carry any other property.

3.2 Modeling Behavior as Paths

Directly-follows (Events) Events are ordered by time from the viewpoint of an entity they are correlated to (cf. Sect. 2.1). As our model allows events to be correlated to multiple entities, each event may have multiple "next" events, depending on the entity. We model that two *:Event* nodes x and y are temporally ordered (from the perspective of an entity) by a *:DF*-relationship from x to y. Each *:DF* relationship goes forward in time and holds the *EntityType* for which this relationship holds as a property. Moreover, the events x and y both have to correlated to the same entity node via *:CORR* relationships.

In Fig. 4, events $e1$, $e2$, and $e9$ are connected by *:DF* relationships from $e1$ to $e2$ and from $e2$ to $e9$ carrying the property *EntityType* = *Application*. Moreover, $e1$, $e2$, and $e9$ are all correlated to entity *Application 1* through the *:CORR* relationships. Thus, the *:DF* relationships form a trace $\langle e1, e2, e9\rangle$ from the perspective of *Application 1*. In the same manner, $\langle e4, e5, e7\rangle$ form a trace from the perspective of entity *Offer 1*, and $\langle e1, e2, e7\rangle$ is a trace from the perspective of *Resource 10*. Note that this trace shares events e_1 and e_2 with the first trace and event e_7 with the second trace. Also, note that the graph contains a second *:DF* relationship from $e1$ to $e2$ with property *EntityType* = *Resource*.

Design decisions for modeling directly-follows Our data model requires that each *:DF* relationship is specific to exactly one entity type. This constraint ensures that queries for behavioral paths can be easily written and efficiently evaluated.

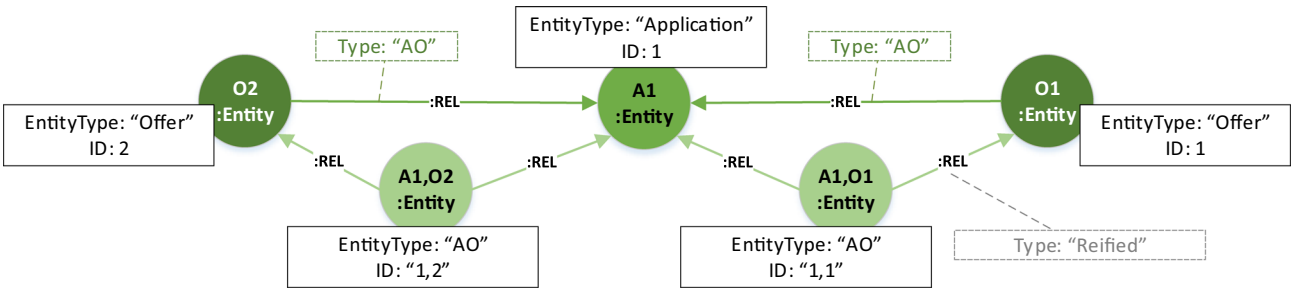


Fig. 5 Graph describing relations between entities Application 1, Offer 1, and Offer 2, and the reification of these relations into composite entities

Suppose we modeled only a single *:DF* relationship between e_1 and e_2 that describes behavior for both *Application 1* and *Resource 10*. If we try to query a path from $e1$ to some other event along one specific entity n (to which e1 is correlated), then we would have to test along each *:DF* relationship that the events are correlated to the same entity n as $e1$. This becomes infeasible when querying multi-entity paths [such as Fig. 1(6)]. Also, the query evaluation becomes inefficient as all possible paths along all reachable *:DF* relationships have to be explored.

The most fine-grained way of modeling would be to require a dedicated *:DF* relationship per entity identifier (with a corresponding *EntityID* property). However, this hinders querying for paths based on entity *types*, e.g., all paths for *Offer* entities.

By modeling one *:DF* relationship per entity type, we enable explicitly querying for behavioral paths per entity types. We explicitly exploit this in query Q6 in Sect. 6. As a consequence, our data model explicitly allows to model multiple behavioral paths through the event data, each path describes a trace from the perspective of one entity. Events can be part of multiple such traces if they are related to multiple entities.

A possible alternative to describing the entity type as a property of the *:DF* relationship is to *refine* the *:DF* label into a set of labels, e.g., *:DF_Application*, *:DF_Offer*, *:DF_Resource*. While this choice simplifies retrieving paths, it requires more case distinction in aggregation queries.

3.3 Modeling Relations and Interactions between Entities

Relations between entities Our data model includes a generic relationship type *:REL* between entities to describe how entities relate to each other. The nature or name of the relation is stored in property *Type*. For example, Fig. 5 describes that *Offer 1* and *Offer 2* are related to *Application 1* through a relation of type *AO*.

Reification of relations Any *:DF* relationship between two events $e1$ and $e2$ requires an *:Entity* to which both $e1$ and $e2$ are correlated via an *:CORR* relationship. This ensures that any behavioral path is "observed" from the perspective of an entity. If we want to analyze the interaction between two related entities, we have to "observe" the behavior from the perspective of the relation between them. For example, to describe how *Application 1* and *Offer 1* interact, we have to observe the behavior from the perspective of the *AO* relation between them [12]. A plain graph-based data model does not allow us to correlate an *:Event* node to the *:REL* relationship between *Application 1* and *Offer 1*. A standard technique in data modeling is to reify the relationship in this case, that is, to introduce a *derived :Entity node* which represents the relationship. We can then correlate events to this *:Entity* node to model interactions between entities [42].

Figure 5 illustrates the reification of the *:REL* relationships between *Offer 1*, *Offer 2*, and *Application 1* in derived entities *A1,O1* and *A1,O2* of type *AO* (the same as the original *:REL* relationship). The derived entities are related to their original entities through *:REL* relationships of type *Reified*. Note that also n-ary relationships can be reified in this way.

Correlating events to reified relations We can describe the interaction between two related entities $E1$ and $E2$ by correlating the events of $E1$ and of $E2$ to the derived entity. For example, in Fig. 6, events $e1$, $e2$, $e9$ are correlated to *A1* and $e6$, $e8$, $e10$ are correlated to *O2*. Together, all these events are correlated to the derived entity *A1,O2*, resulting in the *:DF*-path $\langle e1, e2, e6, e8, e9, e10\rangle$ for entity type *AO*. The path $\langle e1, e2, e4, e5, e7, e9\rangle$ describes the interaction between *A1* and *O1*. Note that Fig. 6 corresponds to Fig. 1(5).

In this way, all directly-follows relations of the events of Table 1 can be modeled correctly and in a single data structure, making this the only data model to satisfy R0 compared to the other data models discussed in Fig. 1.

3.4 Modeling Aggregations of Events and Behavior

Event classes While we assume each event to have the mandatory attribute *Activity*, events can be classified in other ways as well (cf. E6 in Sect. 2.1). In our model of Fig. 3, each event class is described by a node *:Class* defined by a unique *ID* and a *Type* property that is the same for all event classes defined in the same way, e.g., based on the *Activity* attribute

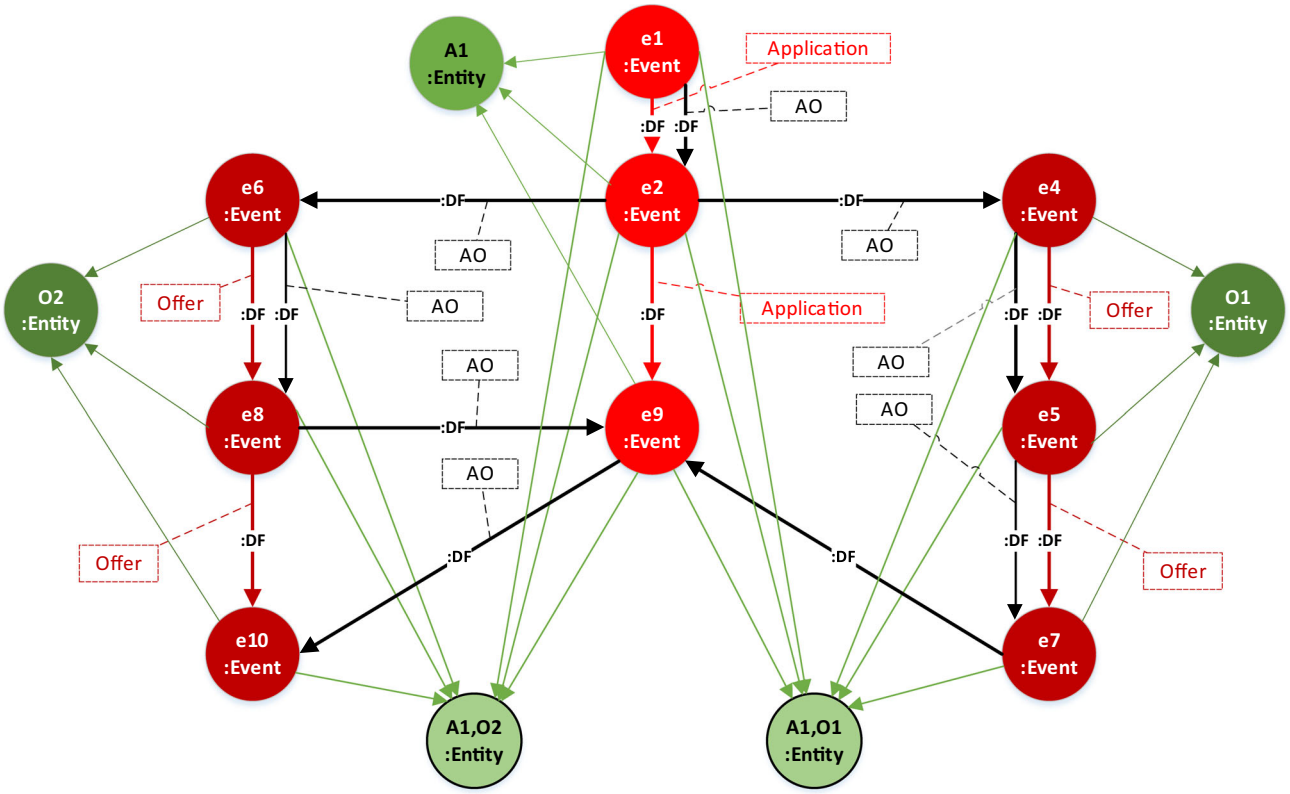


Fig. 6 Graph with additional directly-follows relations along with entities A1,O1 and A1,O2 describes interactions between Application 1, Offer 1, and Offer 2 of Table 1

only or a combination of attributes. Each event can be associated with zero or more event classes by relationships with label *:OBSERVED* from an *:Event* node to a *:Class* node.

Figure 7 shows an example. Two event classes of type "Resource" are defined by the Resource entities occurring in the data, e.g., $Res1$ and $Res2$; events $e6$, $e7$, and $e8$ observed class $Res1$. Further, three event classes of type "Activity" are defined; events $e6$ and $e7$ also observed class A of this type.

Directly-follows on event classes Event data analysis aggregates directly-follows relations between events to directly-follows relations between event classes (cf. Sect. 2.1). Our model of Fig. 3 provides relationship label *:DF_C* between *:Class* nodes. As for *:DF*, each *:DF_C* relationship lists in attribute *EntityType* for which entity type the aggregated directly-follows relationship holds. This allows to describe aggregated behavior per entity type. The graph in Fig. 7 shows the aggregation of *:DF* to *:DF_C*, assuming a single entity type. The two sub-graphs induced by *:DF_C* describe a hand-over of work social network (bottom) [53] and a classical directly-follows graph [54] (top) of the event data in the same model.

4 Semantics of Entities and Events in Labeled Property Graphs

The node and relationship types introduced in Sect. 3 allow us to model multi-dimensional event data in LPGs. However, LPGs allow for unrestricted use of any node and relationship types which would allow for creating LPGs that do not capture the semantics of event data. For example, the LPG in Fig. 8 only uses the node and relationship types of Sect. 3, but the graph violates the semantics the types shall encode: *:DF* does not order events $e2$ and $e3$ according to their timestamp, and events $e1$ and $e2$ are ordered by *:DF* but belong to different entities, and event $e3$ even directly-follows itself.

In this section, we formulate constraints on how the nodes and edges over the types of Sect. 3 may be related, thereby giving them semantics. In the following, we formulate such constraints for any labeled property graph $G = (N, R, label, prop)$ (see Sect. 2.4).

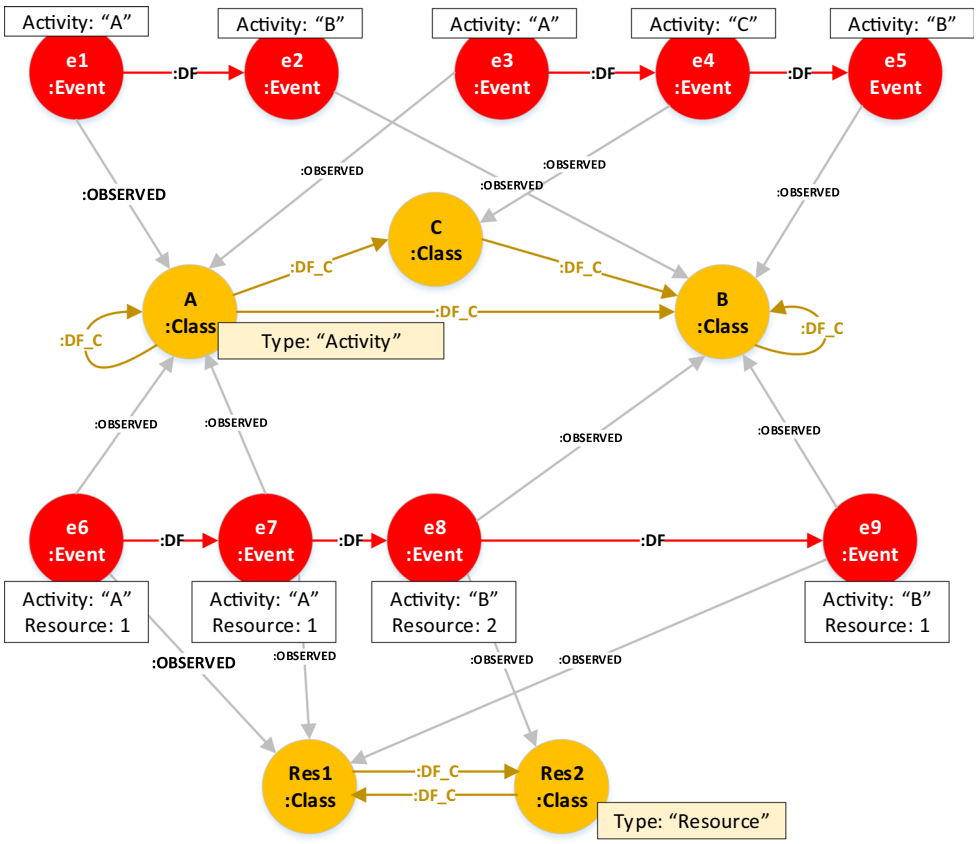


Fig. 7 Graph with multiple event classes and aggregated directly-follows relationships

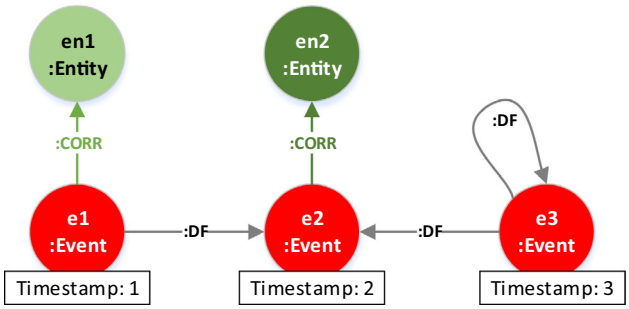


Fig. 8 Incorrect semantic pattern of :CORR and :DF relationships

4.1 Strictly Typed Nodes

We formalize the semantics of the node labels *Entity*, *Event*, *Log*, *Class* and of the relationship labels *REL* (entity to entity), *CORR* (event to entity), *HAS* (event to log), *DF* (directly-follows on events), *OBSERVED* (event to event class), and *DF_C* (directly-follows on event classes).

Each node/relationship may have one of these types (i.e., no node or relationship may have two different semantic roles). Formally, $Label_N = \{Entity, Event, Log, Class\}$, $Label_R = \{REL, CORR, HAS, DF, OBSERVED, DF\_C\}$ and for each $n \in N$, $|label(n) \cap Label_N| \leq 1$ and $|label(n) \cap Label_R| = 0$ and for each $r \in R$, $|label(r) \cap Label_N| = 0$ and $|label(r) \cap Label_R| \leq 1$.

As each node has a unique label in $Label_N$ or $Label_R$, we write $n.label$ and $r.label$ for their labels in the following. Further, we write L for the set $\{n \in N \mid n.label = L\}$ of nodes and of relations $\{r \in R \mid r.label = L\}$ carrying label L, respectively. For example, $n \in Entity$ and $df \in DF$.

4.2 Semantics of Entity–Entity Relations

The Entity–Entity relationship *REL* is a placeholder for all kinds of structural relationships between entities which may carry various semantics. For the scope of this paper, we only require that any *REL* relationship is between two entities, i.e., for any $rel \in REL$, $\overrightarrow{rel} = (n_1, n_2)$ holds $n_1, n_2 \in Entity$.

4.3 Semantics of Event–Entity Relations

The Event–Entity relationship $CORR$ correlates an event to its process entities. While each event e can be related to multiple different entities, for example, an Application and a Resource, there must not be two $CORR$ relationships from one event to the same entity. Furthermore, each event is correlated with some entity, and vice versa, as shown in Fig. 9.

Formally, the following properties have to hold:

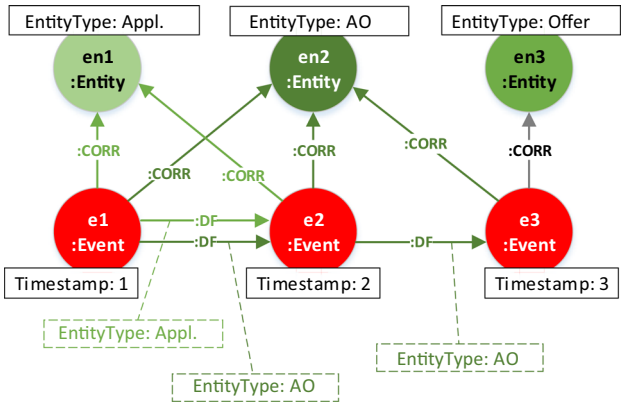


Fig. 9 Correct semantic pattern of :CORR and :DF relationships

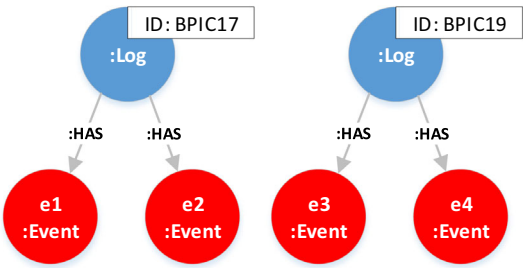


Fig. 10 Correct semantic pattern of :HAS relationships

1. Between any pair of $e \in Event$ and $n \in Entity$ is at most one relation $r \in CORR$, $\overrightarrow{r} = (e, n)$. As a shorthand, we write $CORR \subseteq Event \times Entity$, and $(e, n) \in CORR$.
2. Each event $e \in Event$ is correlated with at least one entity: There exists $(e, n) \in CORR$
3. Each entity $n \in Entity$ is correlated to at least one event: There exists $(e, n) \in CORR$

4.4 Semantics of Log-Event Relations

Log-Event relationship *HAS* explicitly encodes which event belongs to which log. Every event must be in exactly one log, and each log must have at least one event, as shown in Fig. 10. Formally, the following properties have to hold:

1. $HAS \subseteq Log \times Event$
2. Each event $e \in Event$ is in exactly one event log: There exists exactly one $r \in HAS$, $\overrightarrow{r} = (l, e)$.
3. Each event log $l \in Log$ has at least one event: There exists at least one $r \in HAS$, $\overrightarrow{r} = (l, e)$.

4.5 Semantics of Directly-Follows Relation

We model temporal relations as paths of *:DF* relationships over *:Event* nodes. Each *:DF* relationship must go forward in time from the point of view of an *:Entity* node correlated with *both* events involved as shown in Fig. 9. Overall, all *:DF* relationships induce a *partial order*. Formally, the following properties have to hold:

1. For any $df \in DF, \overrightarrow{df} = (e_1, e_2)$ holds $e_1, e_2 \in Event$; note that there can be multiple df relations between the same two events.
2. For every $df \in DF, \overrightarrow{df} = (e_1, e_2)$, exists a log $l \in Log$ with (e_1, l) and $(e_2, l) \in HAS$.
3. For every $df \in DF, \overrightarrow{df} = (e_1, e_2)$, $e_1.Timestamp \leq e_2.Timestamp$ holds, i.e., events are ordered by time.
4. For every $df \in DF, \overrightarrow{df} = (e_1, e_2)$ exists an entity $n \in Entity$ with $(e_1, n), (e_2, n) \in CORR$ such that $n.EntityType = df.EntityType$, and there exists no event e_x correlated to n, $(e_x, n) \in CORR$, such that $e_1.Timestamp < e_x.Timestamp < e_2.Timestamp$ holds, i.e., e_2 directly-follows e_1 from the perspective of entity n.
5. For all events e_1, there is no $df \in DF$ with $\overrightarrow{df} = (e_1, e_1)$, i.e., *:DF* is irreflexive.
6. For all events $e_0, e_n \in Event$ exists no cycle $df_0, \ldots, df_n \in DF$ with $\overrightarrow{df_i} = (e_i, e_{i+1}), i = 1, \ldots, n$, $\overrightarrow{df_n} = (e_n, e_0)$, i.e., *:DF* is acyclic, and hence, the transitive closure of *:DF* is a partial order.

4.6 Semantics of Event-Class relation

Similar to *:CORR*, relationship *:OBSERVED* relates each event to at least one class of the same type, and vice versa. Formally, the following properties have to hold:

1. $OBSERVED \subseteq Event \times Class$
2. Each event $e \in Event$ has at least one event class: There exists $(e, c) \in OBSERVED$, and there are no two classes $(e, c_1), (e, c_2) \in OBSERVED$ with $c_1 \neq c_2$ and $c1.Type = c2.Type$.

4.7 Semantics of Class-level Directly-Follows Relation

The directly-follows relation between *:Class* nodes aggregates existing directly-follows relations between *:Events* nodes wrt. an entity type. The class-level directly-follows relationship *:DF_C* is only defined between *:Class* nodes of the same type. A *:DF_C* relationship relation between two class nodes c_1 and c_2 for an entity type T may only aggregate *:DF* relationships between events that observed c_1 and c_2 and are correlated to the same entity of type T, as shown in Fig. 11. Formally, the following properties have to hold:

1. $DF\_C \subseteq Class \times Class$

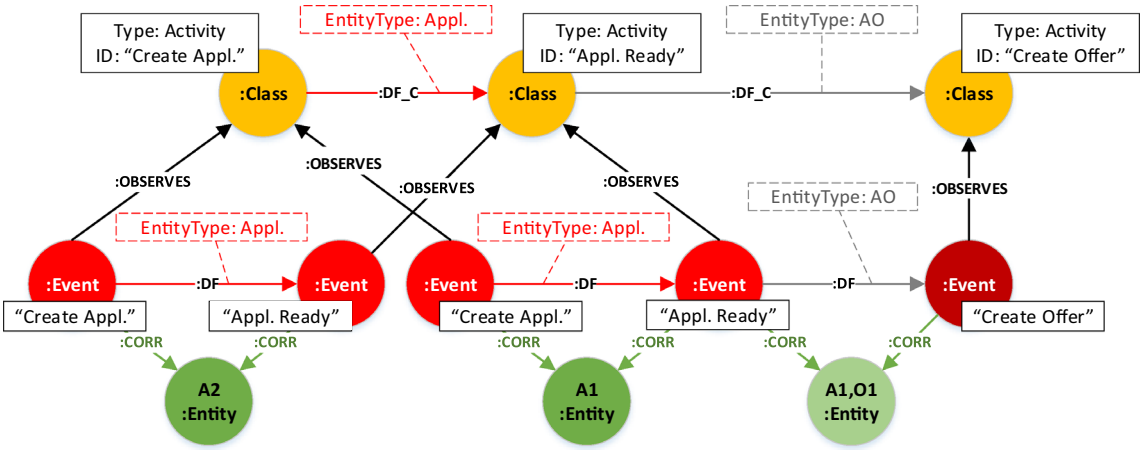


Fig. 11 Correct Semantic Pattern of :OBSERVES and :DF_C Relationship

2. Any two related classes $(c_1, c_2) \in DF\_C$ are of the same type $c_1.Type = c_2.Type$
3. For any two related classes $dfc \in DF\_C, \overrightarrow{dfc} = (c_1, c_2)$ exist events e_1, e_2 of these classes $(e_1, c_1), (e_2, c_2) \in OBSERVED$ ordered in the same way as c_1 and c_2 for the same entity type: There exists $df \in DF, \overrightarrow{df} = (e1, e2)$ and $dfc.EntityType = df.EntityType$.

5 Translating Event Logs to Labeled Property Graphs

We now present a semi-automatic procedure for translating event tables with multiple entity identifiers all stored in a single event table (cf. Sect. 2.3) into the graph data structure introduced in Sect. 3 satisfying the semantic constraints of Sect. 4. This section covers creating the instance-level concepts *:Log*, *:Event*, *:Entity*, *:HAS*, *:DF*, *:CORR*, and *:REL* of Sect. 3, while the type-level concepts *:Class*, *:OBSERVES*, and *DF_C* are covered in Sect. 7.

In a nutshell, our method has the following steps. In Sect. 5.1, we assume the event data to be given in the form of an event table where each record describes one event. In Sect. 5.2, we translate each record with all its attributes to an *:Event* node in the LPG with corresponding properties, obtaining a graph of unrelated *:Event* nodes. In Sect. 5.3, we create *:Log* nodes for each log in the source data set and relate them to the respective *:Event* nodes. In Sect. 5.4, we provide query templates to extract *:Entity* nodes from *:Event* properties (e.g., identifiers) and to correlate *:Event* nodes to all their *:Entity* nodes. In Sect. 5.5, a generic query derives the entity-specific directly-follows *:DF* relationships between events. In Sect. 5.6, we provide query templates to extract *:REL* relationships between entities. In Sect. 5.7, finally, we provide queries to reify relations between entities into derived entities; the queries of step in Sect. 5.5 then infer the *:DF*-relationships for the derived entity allowing to study interactions between entities. We explain the queries on the running example of Table 1.

We demonstrate the types of graphs obtained on the full BPIC17 dataset [28] in Sect. 5.8 and report on a quantitative evaluation of all data sets in Sect. 5.9.

5.1 Source Event Data Format

We expect the event data of the source log to be in event table format (see Sect. 2.3) defining columns *Activity* and *Timestamp* and multiple columns *Attribute1*,…,*AttributeN* that also contain entity identifiers. In case the data comes from multiple logs, also a *LogID* column is required.

5.2 Import the Events

From the source event table (Sect. 5.1), we import each row r as an : Event node e and set for each attribute (column) A of r the property $e.A = r.A$. The following Cypher query implements this step for an event table given as a CSV file.

```
1 LOAD CSV WITH HEADERS FROM "file:///filename.csv" as line
2 CREATE (e:Event {LogID: line.LogID, Activity: line.Activity,
        Timestamp: datetime(line.Timestamp), $Attribute1:
        line.$Attribute1, ..., $AttributeN: line.$AttributeN })
```

Importing the first four rows of Table 1 results in the graph shown in Fig. 12. Importing event table data from other formats than a CSV file requires a corresponding query that applies line 2 of the query to any row in the event table.

5.3 Create Logs

We assume each event e carries attribute $e.LogID$ describing in which log event e was recorded (see Sect. 5.1). For each

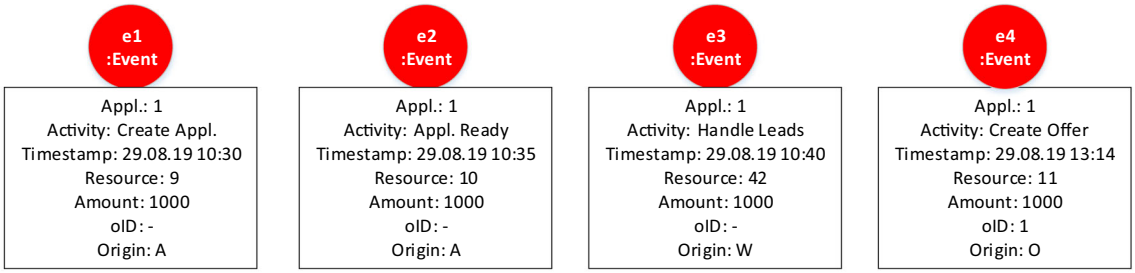


Fig. 12 Graph after step 1: event nodes with properties

unique value $e.LogID$, we create exactly one *:Log* node ℓ with $\ell.ID = e.LogID$. The *MERGE* command in the following Cypher query achieves this.

```
1 MATCH (e:Event)
2 MERGE (:Log {ID: e.LogID})
```

Then, we create a *:HAS* relationship from each event e to the *:Log* node ℓ with $\ell.ID = e.LogID$; the following Cypher query implements this.

```
1 MATCH (e:Event)
2 MATCH (l:Log)
3 WHERE e.LogID = l.ID
4 CREATE (l)-[:HAS]->(e)
```

Applying these queries to the graph of Fig. 12 results in the structure shown at the top of Fig. 13 which conforms to the constraints of Sect. 4.4.

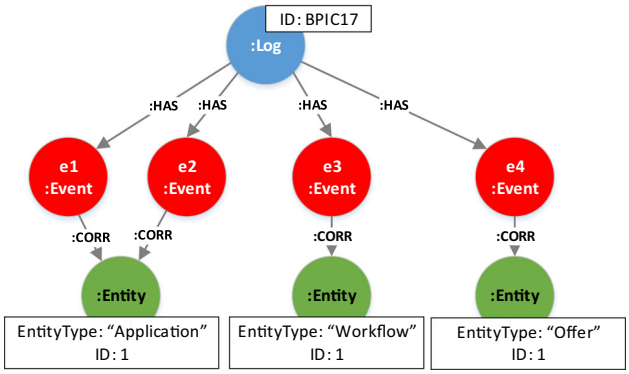


Fig. 13 Graph after creating log and entity nodes

5.4 Create Entities Nodes and Correlate Events (using Domain Knowledge)

We create entities and correlate events to entities in two steps. First, we identify and create the set of all entities that occur in the data by creating *:Entity* nodes. Then, we correlate each event to all its entities by creating *:CORR* relationships.

The general assumption in process event data is that if an event e is correlated to an entity of a specific *Type*, then e has a specific entity identifier attribute $e.EntityID = id$ where id identifies the entity. Recognizing which attribute is an entity identifier and of which entity type requires *domain knowledge*. Primary key identification techniques can be used to identify candidates entity identifiers [19]. However, the same entity identifier attribute may refer to entities of different types [12] so that the actual type can only be inferred from other event attributes. For instance, in our running example, events can have two different entity identifiers: *Appl* and *oID* (see Table 1 or Fig. 12). The property *Origin* distinguishes to which entity type *Appl* or *oID* refers, e.g., only an event e with $e.Origin = \text{“}A\text{”}$ is correlated to an *Application* entity with id $e.Appl$.

Assuming the user identified the *Condition* when an event attribute *EntityID* refers to an entity of type *Type*, then each unique value for $e.EntityID = id$ found in any event e where $e.Condition$ holds indicates the existence of a distinct entity identified by id. We create for each such value id a new *:Entity* node n with $n.id = id$ and $n.EntityType = Type$. The following Cypher query template with 3 parameters (*Condition*, *EntityID*, *Type*) implements this.

```
1 MATCH (e:Event) WHERE Condition
2 WITH e.EntityID AS id, e.Type as name
3 MERGE (en:Entity {ID:id, uID:(name+id), EntityType: name})
```

The query also sets property *uID* as required in Sect. 3. Calling the template with $EntityProperty \equiv \text{“}e.Origin = \text{“}A\text{”}\text{”}$, $EntityID \equiv$ “Appl,” $Type \equiv$ “*Application*” on the graph of Fig. 12 first matches event nodes $e1$ and $e2$ (only these have $e.Origin = \text{“}A\text{”}$) and then returns their value for $e.Appl$, i.e., 1 and 1. Finally, one *:Entity* node n of type Application with $n.ID = 1$ is created.

Now, we can materialize that an event e correlates to an entity n of type *Type* as a *:CORR* relationship from e to n if e satisfies the *Condition* and refers to n by $e.EntityID = n.ID$. The following Cypher query implements this:

```
1 MATCH (e:Event) WHERE Condition
2 MATCH (n:Entity {EntityType: Type}) WHERE e.EntityID = n.ID
3 CREATE (e)-[:CORR]->(n)
```

The above query template has to be executed for each entity type in the data. Assuming each event contains correlation information for at least one entity, it conforms to the semantic requirements of Sect. 4.3. The event graph after applying both queries for Application, Workflow, and Offer entities in this way is shown in Fig. 13.

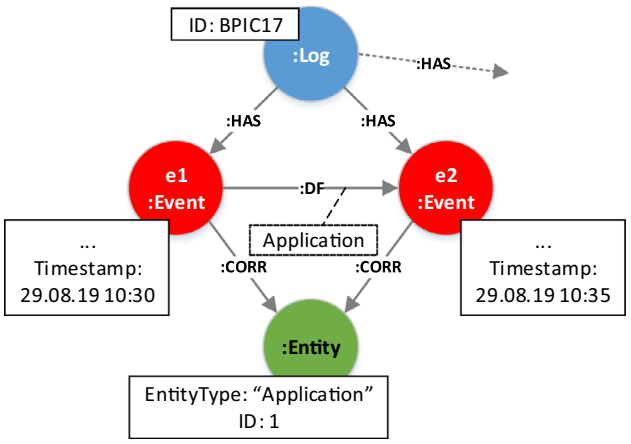


Fig. 14 Graph after adding directly-follows relationships between Application events

5.5 Create Entity-specific Directly-Follows Relation

We now can construct the behavioral *:DF*-relationships from the perspective of each entity n in the graph individually. We retrieve all events $e_1, \ldots, e_k$ correlated to n (via *:CORR*), order them by $e_i.timestamp$, and create a *:DF* relationship from event node e_i to event node e_{i+1}. The following Cypher query implements this: Lines 1-4 collect all events correlated to entity node n in an *eventList* of length k (ordered by their timestamp attribute). We then iterate over the 0-indexed $eventList = \langle e_0, \ldots, e_{k-1} \rangle$ (lines 5-6) and create a *:DF* relationship from e_i to e_{i+1} for each $i = 0, \ldots, k-2$.

```
1 MATCH (n : Entity )
2 MATCH (n) <-[:CORR]- ( ev )
3 WITH n , ev as events ORDER BY ev.timestamp,ID(e)
4 WITH n , collect ( events ) as eventList
5 UNWIND range(0,size(eventList)-2) AS i
6 WITH n , eventList[i] as e1, eventList[i+1] as e2
7 MERGE ( e1 ) -[df:DF {EntityType:n.EntityType}]->( e2 )
```

The data may contain events with identical timestamps, typically due to coarse-grained or imprecise recording [55,56]. To ensure that all directly-follows relations form a directed acyclic graph (see Sect. 4.5), we need to provide a globally consistent ordering for events with identical timestamps. We do so using the internal unique $ID(e)$ of the *:Event* nodes in line 3 to order events by $ID(e)$ in case their timestamps are identical. As we import the events in the same order as in the source data $ID(e)$ is consistent with the implicit ordering in the source data.

The query creates *:DF* relationships for events *per entity* node in the graph; through *MERGE* in line 7, we ensure that we only add relationships between different events per *EntityType* as discussed in Sect. 3.2. Applying the above query on the graph of Fig. 13 results in the *:DF* relationship from $e1$ to $e2$ shown in Fig. 14. Creating *:DF* relationships in this way conforms to the constraints of Sect. 4.5.

5.6 Deriving relations between entities (using domain knowledge)

Entity creation and correlation may leave events of different entities unrelated if an event is not explicitly related to more than on entity. In our running example of BPIC17 [28], events are correlated to either an Application, Offer, or Workflow entity as shown in Fig. 13. Deriving directly-follows relations per entity as in Sect. 5.5 leaves their behaviors disconnected.

We cannot further correlate *Offer* events to other existing entities such as *Application*. If we would correlate $e3$ and $e4$ directly to *Application 1* by entity identifier *Appl.*, we would "pollute" the directly-follows relation of *Application 1* with events that are only remotely related to it, resulting in convergence errors (see Sect. 2.2).

To ensure requirement R0, we extend the data with structural relationships between entities. We can later reify these relationships into entities that describe interactions between other entities. Our data model starts from recorded events, and thus, we have to infer relations between entities from event attributes using domain knowledge. Alternatively, foreign key identification techniques can be used to identify candidates relationships [19].

Assume two events *(e1:Event) -[:CORR]-> (n1:Entity)* and *(e2:Event) -[:CORR]-> (n2:Entity)* are correlated to different entities $n1 <> n2$. If $e2$ contains some property *refto1* referencing the entity identifier *ID* of $n1$, i.e., a foreign key, we observe that $n2$ is related to $n1$ through event $e2$. In our running example, we observe that *Offer 1* is related to *Application 1* through event *e4* of *Offer 1* via property *Appl.*, see Fig. 16.

We lift this observation to entity types. The relation R from entity *type1* to entity *type2* is the set of all pairs $(n1.ID, n2.ID)$ where (1) there is an entity $n1$ of *type1*, (2) some event *(e2:Event) -[:CORR]-> (n2:Entity)* is correlated with entity $n2$ of *type2*, (3) with $e2.refto1 = n1.ID$. Lines 1-4 of the query template below identify such a pair of entities $n1$ and $n2$ for chosen parameters *type1*, *type2*, and *refto1*, and line 5 constructs this relation R of *typeR*.

```
1 MATCH (e1:Event) -[:CORR]-> (n1:Entity) WHERE n1.EntityType='type1'
2 MATCH (e2:Event) -[:CORR]-> (n2:Entity) WHERE n2.EntityType='type2'
3   AND n1 <> n2 AND e2.refto1 = n1.ID
4 WITH DISTINCT n1.ID as n1_id, n2.ID as n2_id
5 WHERE n1_id <> 'Unknown' AND n2_id <> 'Unknown'
6 CREATE (n1) <-[:REL {Type:'typeR'}]- (n2)
```

Applying the above query for *type1* ≡ *Application*, *type2* ≡ *Offer*, *refto1* ≡ *Appl*, *typeR* ≡ *Case_AO*, on our running example results in the relationship of type *Case_AO* from *Offer 1* to *Application 1* shown in Fig. 15.

5.7 Reify relations between entities for describing interactions

To model behavioral relations between two structurally related entities, we reify their relation into a new entity type.

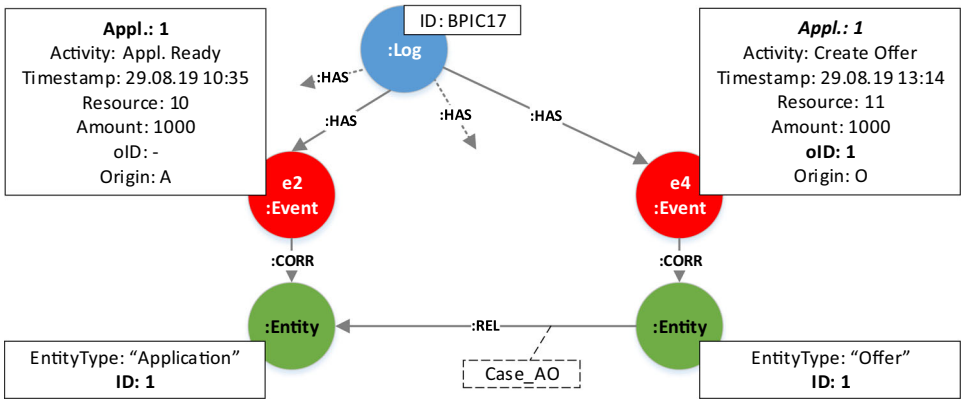


Fig. 15 Graph after inferring the relation between *Application 1* and *Offer 1*

Each pair $(n1, n2)$ of entities in some relation R gets materialized as a new entity r. By correlating the events of $n1$ and $n2$ to r and ordering them over time, we can study the behavior or interaction between $n1$ and $n2$ along r.

The following query reifies each pair $(n1, n2)$ in a *:REL*-relationship of "*typeR*" into a new Entity r and relates r to $n1$ and to $n2$ as relationships of type "Reified."

```
1 MATCH (n1:Entity) -[rel:REL {Type:'typeR'}]-> (n2:Entity)
2 CREATE (n1) <-[:REL {Type:'Reified'}]- ( r : Entity {
        ID:n1.id+'_'+n2.id, EntityType : 'typeR',
        uID:'typeR_'+n1.id+'_'++n2.id) } ) -[:REL {Type: 'Reified'}]->
        (n2)
```

Applying this query on the graph of Fig. 15 for *typeR* ≡ *Case_AO* yields the new entity (1, 1) of type *Case_AO* in Fig. 16 which refers to both original entities *Application 1* and *Offer 1*.

Any event e correlated to an entity n to which the composite entity r of *typeR* refers by a "Reified" relation also correlates with r. The following query achieves this.

```
1 MATCH ( e:Event ) -[:CORR]-> (n:Entity) <-[:REL {Type:"Reified"}]-
        (r:Entity {EntityType: 'typeR'})
2 CREATE (e) -[:CORR]-> (r)
```

Applying this query on the graph obtained in the previous step for *typeR* ≡ *Case_AO* adds the *CORR* relationship from $e2$ and $e4$ to entity (1, 1) of type *Case_AO*. Depending on the available domain knowledge, the correlation query can be made more specific by adding *WHERE* clauses for only correlating events that satisfy specific properties.

We may now derive *:DF* relationships for the composite entity *typeR* using the queries of Sect. 5.5, e.g., Fig. 16 also shows relationship *:DF* for *Case_AO*. By correlating events of related entities *Application 1* and *Order 1* to their own reified entity *Case_AO (1,1)*, we constructed a new *:DF*-relationship for entity type *Case_AO* describing only the interaction between *Application 1* and *Offer 1*. The original directly-follows relations for Application and Offer remain as before and "unpolluted."

5.8 Demonstration on BPIC17

We applied the above queries on the events of the full BPIC17 dataset [28]. After importing all events,[4] we derived entities for 3 types: *Application*, *Workflow*, *Offer*. We reified the binary relations between the three entities into *Case_AO*, *Case_AW*, and *Case_WO* and derived their *:DF* relationships.

Figure 17 shows the graph of handling loan application 681547497 involving one Application entity (dark blue), one Workflow entity (light blue), and two Offer entities (orange). Interactions are shown through the gray *:DF*-relationships of *Case_AO*, *Case_AW*, and *Case_WO*.[5] The graph shows how both Offers are created and handled concurrently to the application entity.

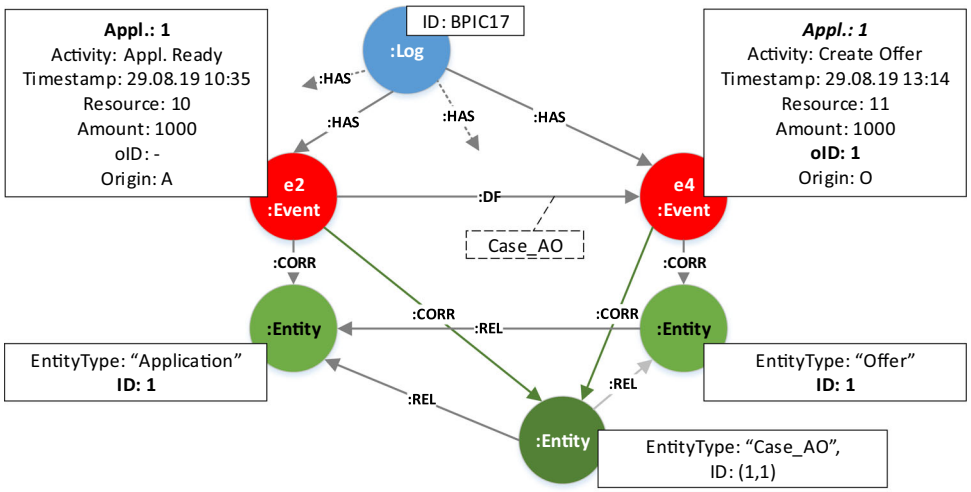
Figure 18 visualizes 7 randomly selected process executions: The 1st and 4th involve only one Offer, whereas all others involve two Offer entities; some executions in BPIC17 involve 5 or more Offer entities. Offers may be created in parallel (2nd, 7th) or with Application events in between (3rd, 5th, 6th). Offers may conclude in parallel (2nd, 3rd, 4th) or with Application events in between (6th, 7th).

We then derived *Resource* as additional entity from the *e.resource* property of events. While Application, Workflow, and Offer are local to a process execution, the *Resource* entities describe works who persist in the system and work on many entities. We derived the *:DF* relationships for *Resource* entities. Querying the data for the events of the 7 process executions of Fig. 18 and the *:DF* relationships of all entities results in the graph in Fig. 19, where *Resource-:DF* relationships are shown in red.

We can clearly see that all process executions and entities are tightly connected through the resources. Each resource is always involved for a sequence of several events of the same

[4] For the visualizations in this section, we filtered out events with lifecycle attribute *suspend* or *resume* for reducing the size of the figures.

[5] To simplify the visualization, the graph does not contain *:DF_Case_AO*, *:DF_Case_AW*, *:DF_Case_WO* relationships which are in parallel to a *DF_Application*, *DF_Workflow*, *DF_Offer* relationship.



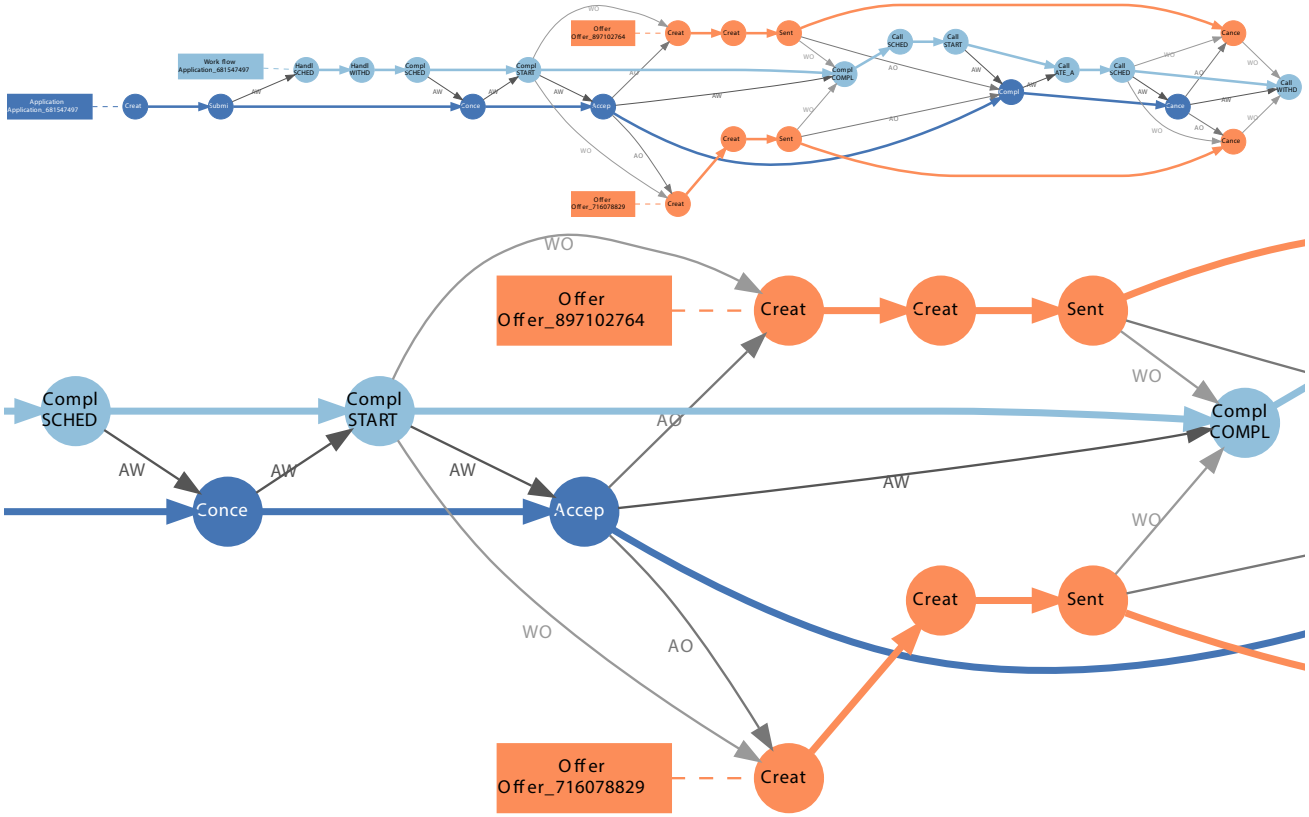


Fig. 16 Graph after reifying relation between *Application 1* and *Offer 1* into composite entity *Case_AO (1,1)*

Fig. 17 Graph of handling loan application 681547497 in BPIC17 [28] (top) with detail of two parallel offers (bottom)

or related entities and then moves to another entity in another process execution while handing the previous entity over to another resource.

Overlaying the *Resource-:DF* relationships on the graphs also allows us to see that interactions between related Application, Workflow, and Offer entities of the same process execution are not explained by Resource entities. In the graph in Fig. 20, the first event of *Offer_1647347263* correlated with *User_85* follows after an Application event (via *Case_AO*) correlated to *User_7*, i.e., there is no resource explaining the ordering of Application and Offer. This confirms the importance of reifying relations between entities into composite entities–otherwise, the graph would describe that *Offer_1647347263* would start concurrently to all preceding events.

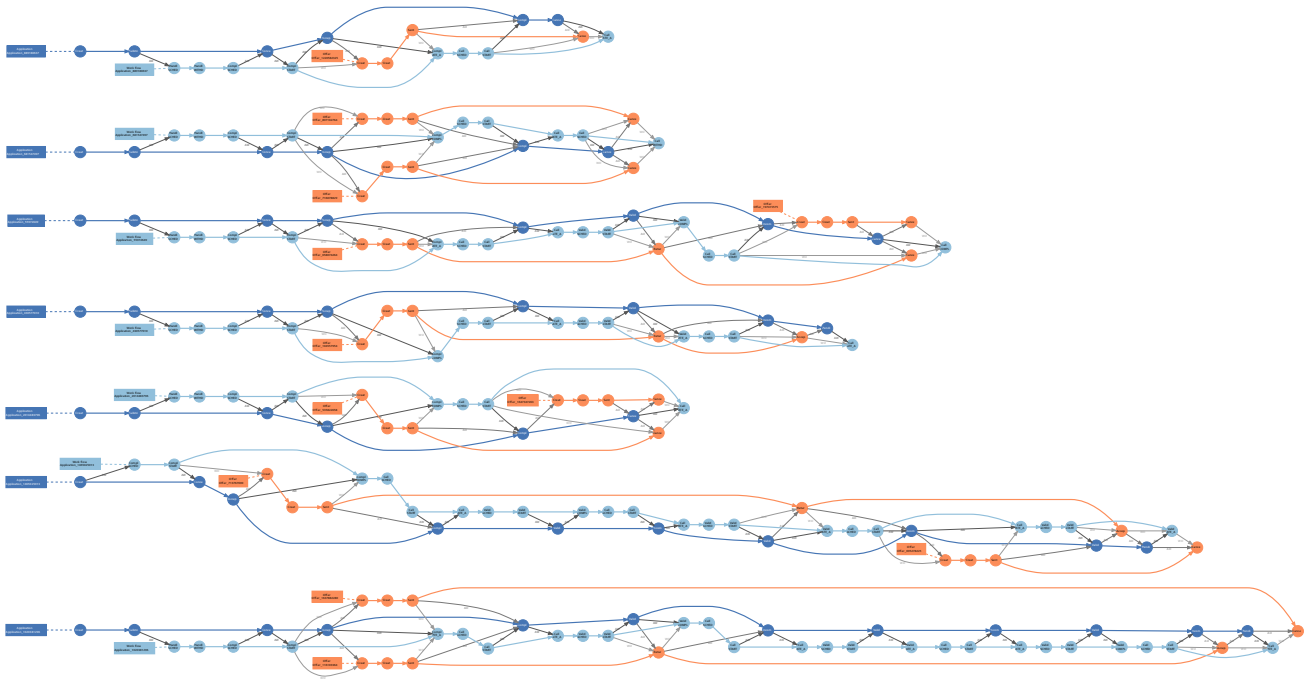


Fig. 18 Graph of 7 randomly selected process executions of BPIC17 [28]

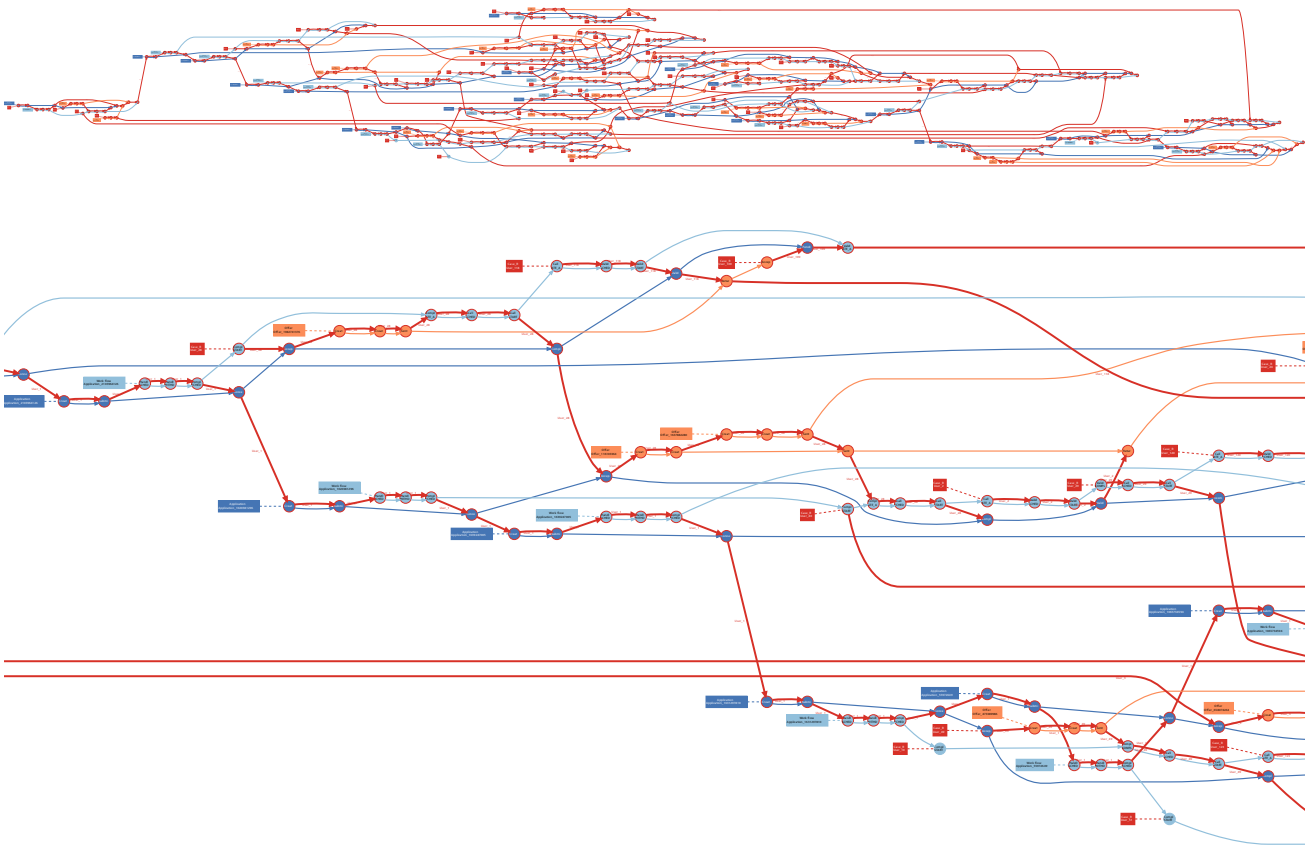


Fig. 19 Behavior of resources overlaid on the 7 process executions of Fig. 18(top) and some details (bottom)

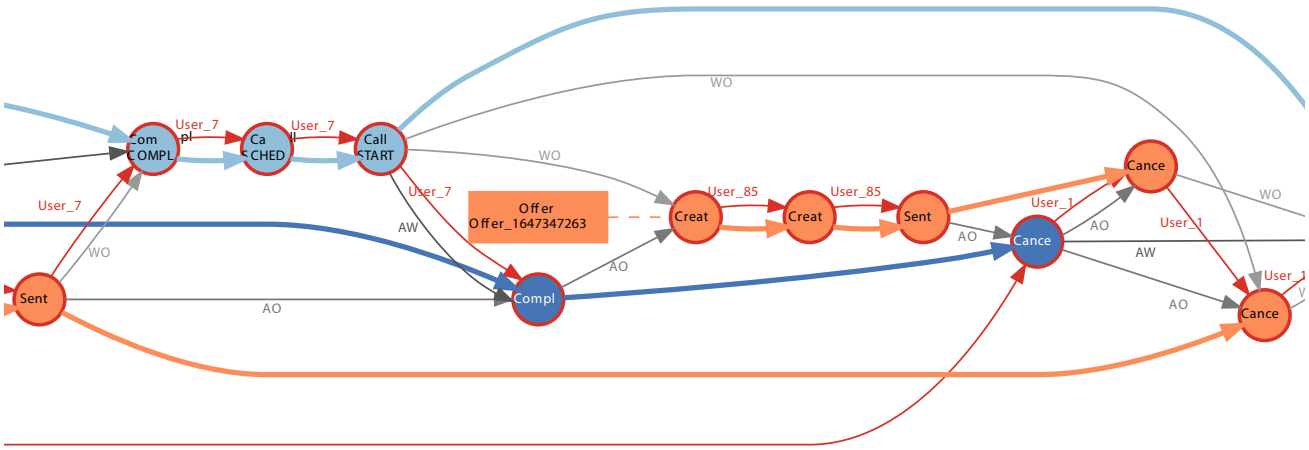


Fig. 20 Resource behavior does not explain entity interactions

5.9 Evaluation

We applied the above steps for importing and transforming the event data into our proposed graph-based data model on 5 real-life datasets [26–28,30,31][6] using a Neo4j instance with 20GB of main memory allocated. The Cypher queries are available at https://github.com/multi-dimensional-process-mining/graphdb-eventlogs and at [41]. The resulting graphs are available at [34–38].

Table 2 summarizes the domain knowledge we applied for each data set during the conversion in terms of the number of distinct event logs, entity types, relationship types between entities, and how many relationship types were materialized as derived entities. Although the source data for BPIC'14 [26] and BPIC'16 [27] contains several CSV files, the event records in these CSV files refer to shared entities. We therefore did not import the events from these files under separate *:Log* nodes but *integrated* them under a single *:Log* node. Only BPIC'15 [31] contains 5 distinct event logs which were also imported as separate *:Log* nodes. Each data set contained multiple entity types. All data sets contain a resource entity type. BPIC'14 [34] has no classical case identifier at all but describes a complex interplay of hard- and software *Components* and their *Configuration Items*, for which *Incidents* are reported which are handled in *Interactions* leading to *Changes* on the *Components* supported by *Knowledge Base Entries*. BPIC'15 [35] has a classical case notion and *3 different notions of Resources* involved in an event. BPIC'16 [27] has a *Customer*, *Complaints* which are collected in *Dossiers*, users seeking information on a website in *Browsing Sessions* from an *IP Address*. BPIC'17 [37] has *Loan Applications*, *Offers*, and a *Workflow* instance handling these documents, and entities derived from the relations between these. BPIC'19 [38] has *Purchase Orders* containing *Items* supplied by *Vendors*. Only BPIC'17 required materializing relationships to represent the data correctly.

Table 2 Type information of labeled property graphs of 5 real-life event data sets

Data Set	:Log nodes	:Entity types	:REL types	:REL types materialized
BPIC'14 [34]	1	7	2	0
BPIC'15 [35]	5	4	2	0
BPIC'16 [36]	1	7	0	0
BPIC'17 [37]	1	7	3	3
BPIC'19 [38]	1	4	1	0

All data sets could be converted using our method. We measured the size of the resulting graphs, the memory requirements for storing the data in Neo4j, and the time required for the conversion. Table 3 shows the number of nodes and relationships for the labels *:Event*, *:Entity*, *:Class*, *:DF*, *:CORR*, and *:REL*. Where a classical event log of n events and m cases only contains $n - m$ *:DF*-relationships, all event graphs have between $1.9 \cdot n$ (BPIC'17) and $4.8 \cdot n$ (BPIC'16) *:DF*-relationships due to the additional entity types (c.f. Tab 2). Consequently, almost all events have more than one incoming or outgoing *:DF*-relationship. On average, in each graph, each event is related to four other entities (c.f., *:CORR* vs *:Entity*). By comparison, structural relations are much sparser in the graphs. Further, in each graph, each event has one *:HAS* and one *:OBSERVES* relationship. Overall, the graphs have between 7.2 (BPIC'14, BPIC'17) and 10.7 (BPIC'16) edges per node. Figure 21 illustrates the typical structures found in these graphs by a (small) part of the BPIC'14 graph [34]. The graph shows 17 entities of 5 different types that are all directly or indirectly connected.

[6] For BPIC2016, we omitted all click events without a session identifier as these could not be correlated.

Table 3 Logical size of labeled property graphs of 5 real-life event datasets

Data Set	Nodes			Relationships		
	:Event	:Entity	:Class	:DF	:CORR	:REL
BPIC'14 [34]	690,622	228,885	330	2,503,328	2,732,213	65,601
BPIC'15 [35]	262,628	5,649	356	1,044,650	1,050,512	107
BPIC'16 [36]	7,360,146	26,647	620	35,681,967	36,430,880	0
BPIC'17 [37]	1,202,267	255,170	66	2,298,372	5,244,794	352,497
BPIC'19 [38]	1,595,923	328,083	52	5,653,917	5,984,602	251,734

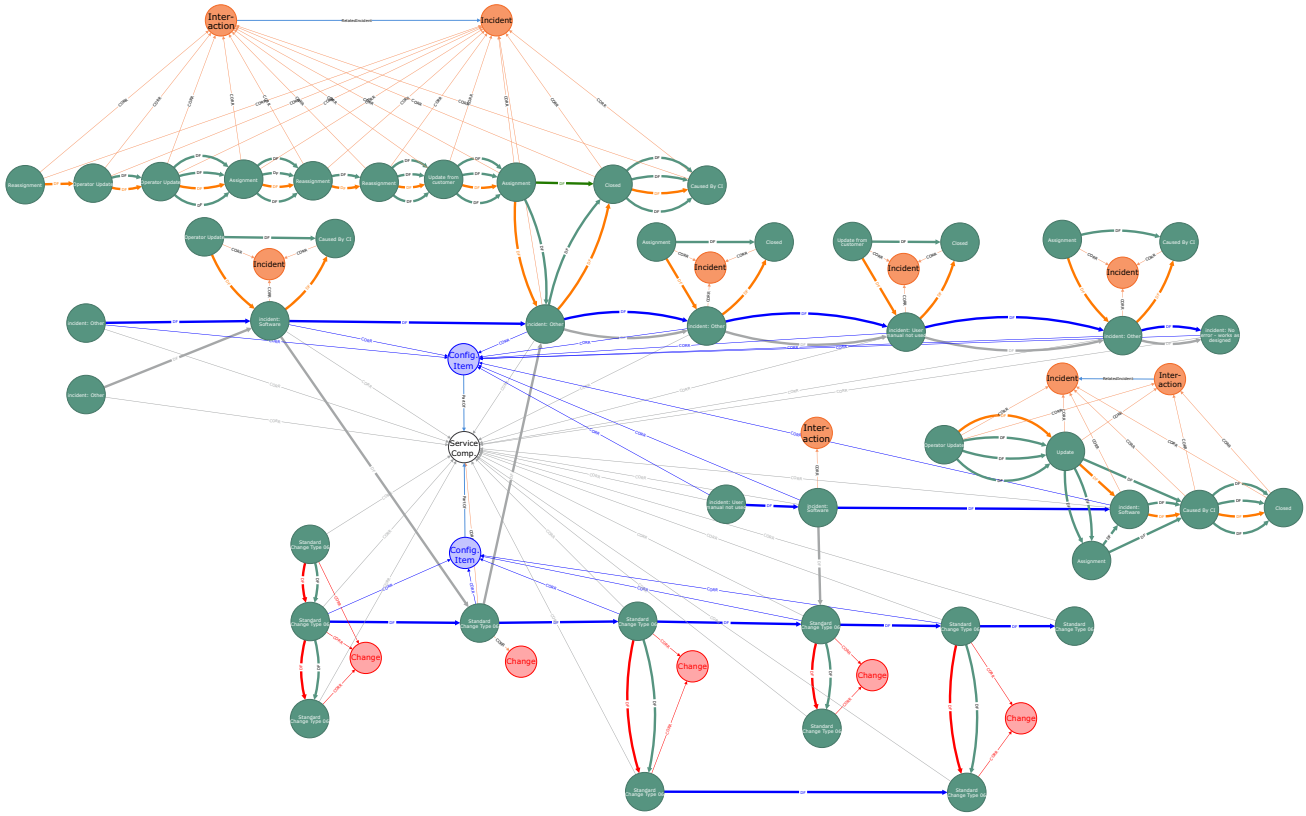


Fig. 21 Part of BPIC'14 data showing a service component (white) with two related configuration items (blue), and related incidents and interactions (orange) and changes (red) and their events (green)

Table 4 lists the resulting storage sizes of the Neo4J instances and conversion times. Explicitly encoding the structural and behavioral information in the graph requires a factor 3 (BPIC'15) to 13.3 (BPIC'17) more space than the source data in CSV format. Relationships account for 35%-50% of the space, while properties account for 40%-61% of the Neo4j storage.

All translations except for BPIC'15 [31] succeeded within several minutes. We observed execution time to depend on two particulars. (1) On the product $n \cdot k$ of the number n of events and k of entity types: Our conversion approach iterates over all n events at most $2 \cdot k$ times (to create an entity from an event attribute and to derive the *:DF* relationship for this entity). The large number of events and entity types thus causes the higher running time for BPIC'15. (2) On the number m of entity nodes when deriving structural relationships, in the worst case (almost) all m^2 pairs of entities nodes have to be checked for the presence of a relation, which was the case in BPIC'15 and for BPIC'14, where execution time without constructing *:REL* relationships is significantly lower.

In general, the size of the source log is not a reliable indicator for the size of a graph event log. For example, the BPIC'14 source data [26] is small in size but defines several related entities resulting in a large graph. More importantly, any graph can be adapted to the needs of a particular research question, e.g., by deriving only a limited number of (composite) entities.

Table 4 Resulting database size and conversion time for converting 5 real-life event datasets into labeled property graphs

Data set	Source size	DB size (GB)					Time (mins)	
	(GB)	Nodes	Prop.	Strings	Rel's	Total	All	No :REL
BPIC'14 [34]	0.08	0.01	0.34	<0.01	0.31	0.67	11.2	3.1
BPIC'15 [35]	0.11	<0.01	0.20	0.01	0.12	0.33	11.7	1.3
BPIC'16 [36]	1.06	0.12	3.42	0.76	4.27	8.60	58.6	58.6
BPIC'17 [37]	0.29	0.80	2.37	0.05	1.36	3.87	8.2	8.0
BPIC'19 [38]	0.52	0.02	0.96	0.24	0.90	2.13	9.7	8.7

6 Querying Multi-Dimensional Event Data

In the following, we present 6 classes of analysis questions that we formulated to evaluate requirements R5–R11 of Sect. 2.2 for querying multi-dimensional event data on the LPGs of Sect. 5. In Sect. 7, we evaluate the aggregation requirements R12–R15.

We conducted the querying experiment on the BPIC'17 dataset for which we additionally derived the *Case_AWO* entity type which corresponds to the original case notion, i.e., all events sharing the same *e.case* attribute. We did this in order to be able to verify the correctness of our results against classical process mining software which works with the original case notion only. For each analysis question, we provide a Cypher query and report results and the query processing times (measured on a 6 Core Intel i7-9850H @ 2.60 GHz windows machine with 32 GB RAM @ 2667 MHz with Neo4j Browser).

6.1 Graph Queries

Q1. Query attributes of events/cases We want to query for the first-class concepts of event logs: a case and an event based on event/case attributes by using a partial patterns to satisfy R7. The following query returns the event attribute "timestamp" and the case attribute "LoanGoal" of Case "Application_681547497".[7] Note that all (event and entity) attributes are encoded as properties of event nodes.

```
1 MATCH (c:Entity {EntityType: 'Case_AWO'}) <-[:CORR]- (e:Event)
2 WHERE c.ID = "Application_681547497" AND e.Activity = "A_Submitted"
3 RETURN e.timestamp, e.LoanGoal
```

The query has been processed in 0.079 s. After modifying the query to consider all cases, i.e., remove the condition for a specific case in line 2, the query completed in 0.117 s.

Q2. Query directly-follows relations Q2 is focused on temporal aspects. Here, we show a query that satisfies R8 by considering 2 consecutive events. Directly-follows relations of events in a case are an important characteristic of event logs as they represent the case internal temporal order of events and many of today's process mining techniques rely on these relations. The query below returns the event directly following the node with the activity property "O_Created" of a given offer entity by matching the *:DF_Offer* relationship.

```
1 MATCH (o:Entity {EntityType: 'Offer'}) <-[:CORR]- (e1:Event) <-[:DF
      {EntityType: 'Offer'}]- (e2:Event)
2 WHERE o.ID = "Offer_716078829" AND e1.Activity = "O_Created"
3 RETURN e1,e2
```

The query execution time for one specific offer was 0.160 s, whereas querying the *:DF_Offer* relations with destination node "O_Created" for all 42,995 offers took 0.146 s. Directly-follows relations of other entities (Application and Workflow) or across entities (Case_AWO) can be queried by adjusting the query in the *MATCH* and *WHERE* clauses accordingly.

Q3. Query eventually-follows relations We want a query that satisfies R8 by considering the temporal relationship of any 2 events of a case. Eventually-follows relations are also related to the case internal order of events. Event y *eventually-follows* event x if y occurs after x in the same case, that is, if x and y are connected through a path of directly-follows relations of arbitrary length. We query the offer specific eventually-follows relationship between "O_Created" and "O_Cancelled" for a given offer as follows:

```
1 MATCH (o:Entity {EntityType: 'Offer'}) <-[:CORR]- (e1:Event) -[:DF*
      {EntityType: 'Offer'}]-> (e2:Event)
2 WHERE o.ID = "Offer_716078829" AND e1.Activity = "O_Created" AND
      e2.Activity = "O_Cancelled"
3 RETURN e1,e2
```

Even though the *MATCH* clause looks similar to the one of the directly-follows query, the *-Operator changes the pattern from a direct relationship to a path of arbitrary length. Since we want to find the eventually-follows relationship of two specific activities, we also added condition $e2.Activity =$ "*O_Cancelled*" to the *WHERE* clause to define the endpoint of the paths we want to match in the graph. For the given offer, the query took 0.161 s. For all 20,898 offers where "O_Created" is eventually followed by "O_Cancelled," we removed the condition for "Offer_716078829" from the query which then took 0.140 s.

Q4. Case variants We want a query to return a case variant as path in the graph to satisfy R6. A *case variant* is the sequence of activities of a case. Case variants are, for example, used to detect frequent behavior of a process. We can query the

[7] The specific cases were randomly selected from all available cases in the data.

graph to retain the path of events of a case by walking over all of its directly-follows relationships from the first to the last event. For a given case (Case_AWO), this can be done as follows:

```
MATCH (c:Entity {EntityType: 'Case_AWO'}) <-[:CORR]- (e1:Event)
      -[:DF* {EntityType: 'Case_AWO'}]-> (e2:Event)
WHERE NOT ()-[:DF {EntityType: 'Case_AWO'}]->(e1) AND NOT (e2)-[:DF
      {EntityType: 'Case_AWO'}]->() AND c.ID = 'Application_681547497'
RETURN (e1:Event) -[:DF* {EntityType: 'Case_AWO'}]-> (e2:Event) AS
      paths
```

The pattern of the match clause follows the same logic as the eventually-follows match pattern. For variants, we limit the output to the first and last event of a case, i.e., the events that have no incoming or no outgoing ":DF" relationship with "Case_AWO" entity type. The query completed in 0.250 s. Similarly, we can query the graph for variants of another entity such as Offer. The paths of events returned by the above query can be turned in a list of activity sequences by Cypher's list operators: *UNWIND* processes each path in the *paths* variable iteratively, function *nodes()* translates the path into a list of nodes, and list comprehension maps each event node to its activity property. The resulting list of activities can be compared for equality with other lists, etc.

Q5. Query duration/distance between two specific activities The information on how much time or how many activities were needed to get an Offer from "O_Created" to "O_Accepted," for example, can be used to measure process performance. For Q5, we want to query temporal relations in the form of durations and path lengths to satisfy R8. Say we are interested in the offer entity that took the longest time to get accepted. We can query the eventually-follows relation of two given activities and use their timestamps to calculate the elapsed time between them:

```
MATCH (e1:Event) -[:CORR]-> (o:Entity {EntityType: 'Offer'})
      <-[:CORR]- (e2:Event)
WHERE e1.Activity = "O_Created" and e2.Activity = "O_Accepted"
WITH e1, e2, duration.indays(e1.timestamp, e2.timestamp) AS days,
      duration.between(e1.timestamp, e2.timestamp) AS time, o
ORDER BY time DESC
LIMIT 1
RETURN e1, e2, days.days as days, time.hours  AS hours, time.minutes
      % 60 as minutes , o
```

The query matches all *:CORR* relationships filters for the given activities and then uses Cypher's duration function to calculate the time spans. Only the result with the longest duration is returned. In case we want to retrieve the distance wrt. the number of activities, we can aggregate over the nodes along the path between the two events with eventually-follows relation and count the hops with the "Length()" function as shown in [52]. The query for the elapsed time completed in 1.072 s. Querying for the longest path took 2.176 s.

Q6. Query for behavior across multi-instance relations Event logs such as BPIC'17 can contain multiple case identifiers. A case identifier may be a single entity, e.g., Offer, or any combination of entities such as the Case notion of BPIC'17 combining Application, Workflow, and Offer entities. Querying the behavior across different instances of these entities typically requires multiple steps with traditional event logs such as custom scripts to be able to select, project, aggregate, and combine the results accordingly. With Q6, we want to satisfy R9 by querying for events correlated with the same entity, R10 by combining data from different entities in the same query, and to satisfy R11 by querying 2 (sub)processes in a single query. We defined a query that returns (1) all paths from "A_Create Application" to "O_Cancelled" (2) for only those BPIC'17 Cases that have more than one Offer with "O_Created" directly followed by "O_Cancelled" (from the perspective of the Offer entity, not the global case entity).

```
1 MATCH (o:Entity {EntityType: "Offer"})<-[:CORR]-(e1:Event {Activity:
      "O_Created"}) -[df:DF {EntityType: "Offer"}]-> (e2:Event
      {Activity: "O_Cancelled"})-[:CORR]->(o)
2 MATCH (e2)-[:CORR]->(c:Entity {EntityType:
      "Case_AWO"})<-[:CORR]-(e1)-[:CORR]->(o)
3 WITH c, count(o) AS ct
4 WHERE ct > 1
5 MATCH (o:Entity {EntityType: "Offer"})<-[:CORR]-(e1:Event {Activity:
      "O_Created"}) -[df:DF {EntityType: "Offer"}]-> (e2:Event
      {Activity: "O_Cancelled"})-[:CORR]->(o)
6 MATCH (e2)-[:CORR]->(c)<-[:CORR]-(e1)-[:CORR]->(o)
7 WITH e2 AS O_Cancelled,c
8 MATCH (A_Created:Event {Activity: "A_Create
      Application"})-[:CORR]->(c)<-[:CORR]-(O_Cancelled:Event
      {Activity: "O_Cancelled"})
9 MATCH p = (A_Created) -[:DF* {EntityType: "Case_AWO"}]-> (O_Cancelled)
10 RETURN p
```

The query demonstrates several unique aspects of querying multi-dimensional event data in labeled property graphs.

The first *MATCH* clauses (lines 1-4) return all entity nodes *c* of the original case notion "Case_AWO" which are related to more than one Offer *o* (via *:CORR*) for which "O_Created" is directly followed by "O_Cancelled" (via *:DF* for entity type "Offer"). The second pair of *MATCH* clauses (lines 5-7) return all "O_Cancelled" events that directly succeed "O_Created" (via *:DF* for "Offer") and are correlated to one of the cases *c* with multiple offers (found in Lines 1-4). The returned "O_Cancelled" events are used in the last pair of *MATCH* clauses (lines 8-9) to return paths from some "A_Create Application" event (also correlated to *c*) to one of the "O_Cancelled" events returned previously. This way we get a unique path for every Offer that meets the criteria.

The query's execution time was 1.246 s in Neo4j Browser. Figure 22 shows 2 of the 218 paths of the query's output in Neo4j's graphical representation. The thick black edges show the returned path of events along the *:DF*-relationships for *Case_AWO*. The gray edges show additional *:DF*-relationships for *Application*, *Offer*, and *Workflow* entities, revealing how the path walks across different entities. The red edges highlight where "O_Create Offer" is directly followed by "O_Created," which is the graph feature used by the query to identify these paths.

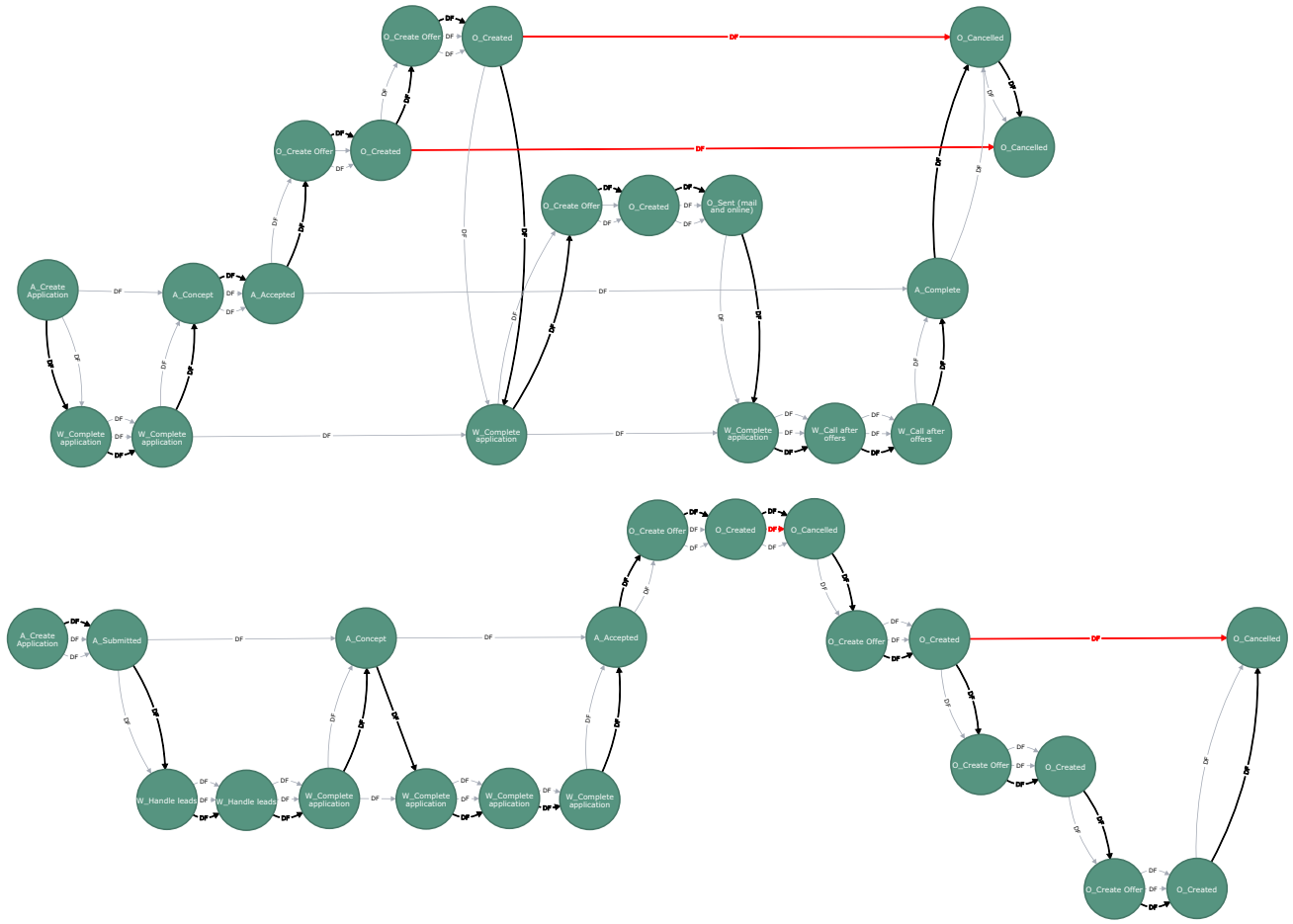


Fig. 22 Q6 output

6.2 Graph Query Validation and Performance

We discuss the validity and performance of our Cypher queries against other implementations.

To validate the above queries again ground truth, we used process mining software such as ProM and Disco with standard operators as baseline implementation to filter the event data on BPIC'17 event log to generate the set of expected results for the graph queries. We used Disco's *Attribute* filter for Q1, the *Follower* filter for Q2 and Q3, the *Attribute* filter on a case attribute for Q4, and the *Follower* filter with constraints on the time between eventually following events for Q5. Q6 could not be queried with classical process mining tools but required a custom procedural algorithm using a single-pass search over the data.

Our Cypher queries obtained the same result as the baseline implementations. The graph analysis for Q1–Q6 required only Cypher queries with clauses and functions as described in [51] (except for the typecasts which are not part of Cypher but provided by Neo4j). A technical report provides full details [52].

We compare the performance of querying our event data model to Disco and to the specialized business intelligence software Qlik Sense, which provides a custom data engine specialized for efficient data analysis. In Neo4j, we used the property graph database as described in Sect. 5 and took the reported query time from the database. The running time in Disco has been measured manually with a preloaded log and a pre-configured set of filtering operations. As Qlik Sense has no native query language, we built a custom data model to transform and filter the data according to the query logic. Populating the data model is included in the measured runtime of the Qlik Sense queries which was measured using system time. We validated the correctness of the Qlik Sense queries against the ground truth developed in Disco.

Table 5 gives an overview of the corresponding results for BPIC'17. All approaches returned the same result. Neo4j was fastest in all queries except Q5 where Disco performed best. Our custom procedural algorithm to validate the correctness of Q6 required 15mins compared to less than 2 s for Neo4j.

Table 5 Query performance and correctness overview

Query	Runtime Neo4j	Runtime Qlik	Runtime Disco	Results Match
Q1 single event	0.079 s	0.571 s	1.10 s	yes
Q1 full log	0.117 s	0.516 s	1.62 s	yes
Q2 single event	0.160 s	1.892 s	1.27 s	yes
Q2 full log	0.146 s	5.866 s	3.28 s	yes
Q3 single event	0.161 s	3.203 s	1.40 s	yes
Q3 full log	0.140 s	3.820 s	2.82 s	yes
Q4	0.250 s	2.095 s	0.89 s	yes
Q5 time	1.072 s	1.679 s	2.34 s	yes
Q5 path	2.086 s	6.391 s	1.54 s	yes
Q6	1.246 s	6.816 s	15 mins*	yes

*Custom script

6.3 Expressive Power of Cypher for Behavioral Queries

We finally discuss the expressive power of Cypher for behavioral queries. We specifically focus on Cypher's abilities to specify paths compared to theoretical graph query languages [57]] and to PQL [39,40].

Regular path queries (RPQs) [57] are queries built from edges (as atoms), edge inversion, path concatenation (sequence), choice between two alternative paths, and the transitive closure. Conjunctive RPQs allow to query for paths which must overlap in named edges or nodes, e.g., the same start and end nodes. Cypher supports path conjunction but not the full extent of RPQs [Ch.3][57] as not all forms of transitive closure are supported. This renders Cypher weaker than regular languages. However, ongoing standardization efforts subsume Cypher in the more expressive graph query language G-Core [58]. Any developments in this field will be applicable to our data model on standard labeled property graphs.

PQL has two alternative approaches to behavioral querying. Scenario-based queries allow to specify a trace pattern with wildcards that has to be matched [39]; the SQL-like syntax of PQL also allows to specify attributes. This renders PQL comparable to behavioral querying with Cypher, though Cypher's conjunctive path queries are more expressive than sequences with wildcards. PQL also supports querying for behavioral relations [40] from the 4C spectrum [59]. In terms of our data model, these behavioral relations make statements over *Class* nodes, i.e., sets of events, for example, "all events of activity A precede all events of activity B." Some behavioral relations of PQL can be expressed in Cypher. For instance, the following query returns all entities of type *X* (process executions) where activities *A* and *B* are in conflict (never occur together).

```
1 MATCH (n:Entity {EntityType:"X"})
2 WHERE (n)<-[:CORR]-(:Event {Activity:"A"}) AND NOT
      (n)<-[:CORR]-(:Event {Activity:"B"})
3 OR NOT (n)<-[:CORR]-(:Event {Activity:"A"}) AND (n)<-[:CORR]-(:Event
      {Activity:"B"})
4 RETURN n
```

However, a more detailed comparison requires further research for two reasons: (1) PQL is aimed at retrieving models which specify a language, while our Cypher-based query language is aimed at retrieving individual executions (words of a language). (2) Our current data model does not express concurrency within an entity, i.e., all events are ordered over time, so concurrency only emerges between different entities. PQL in turn is not designed to handle multiple entities in a process, hindering a comparison.

7 Constructing Simple Models for Multiple Entities

We now show that our data model of Sect. 3 allows *aggregating* the directly-follows relations between events to directly-follows relations between event classes–taking the notion of entities and entity types into account. We provide queries that satisfy R12–R15 of Sect. 2.2.

7.1 Aggregating events into user-defined event classes

An event *:Class* is a node describing a *set* of events with the same characteristics, e.g., having the same *Activity* or other combination of data attributes. We can aggregate events into user-defined event classes using the same principles as deriving and correlating *:Entity* nodes: We query for all distinct values of a particular (combination of) event attributes and create a new *:Class* node per retrieved value(s). The following two queries illustrate the concept for two types of event classes: the combination of activity name and life-cycle attribute, and the resource attribute.

```
1 MATCH (e:Event) WITH distinct e.Activity AS actName,e.lifecycle AS
      lifecycle
```

```
2 MERGE ( c : Class { Name:actName, Lifecycle:lifecycle,
      Type:"Activity+Lifecycle", ID: actName+"+"+lifecycle})
```

```
1 MATCH ( e : Event ) WITH distinct e.resource AS name
2 MERGE ( c : Class { Type:"Resource", ID: name})
```

We then link each *:Class* node to all events of this class when they match on the defining attributes, as for correlating events to entities. We show the query for *Activity+Lifecycle*.

```
1 MATCH ( c : Class ) WHERE c.Type = "Activity+Lifecycle"
2 MATCH ( e : Event ) where e.Activity = c.Name AND e.lifecycle =
      c.Lifecycle
3 CREATE ( e ) -[:OBSERVED]-> ( c )
```

We may also derive event classes based on behavioral properties of events, e.g., based on an event *(e2:Event) -[:DF]-> (e)* preceding *e*. The above queries satisfy the semantic constraints for *:OBSERVED* of Sect. 4.6.

7.2 Aggregating directly-follows relations

The *(c1:Class) -[:DF_C]-> (c2:Class)* directly-follows relation between class $c1$ and class $c2$ aggregates all directly-follows relations *(e1:Event) -[:DF]-> (e2:Event)* between events $e1$ of $c1$ and $e2$ of $c2$, see Sect. 4.7.

We may only aggregate *:DF* relationships between events correlated to the same entity n (line 2) for which the *:DF* relationship was also defined (line 3). Further classes $c1$ and $c2$ must be of the same *Type* (line 3). This ensures that we satisfy R15. We aggregate by counting how many *:DF* relationships (df) exist between $c1$ and $c2$ (line 4) and create a *:DF_C* relationship for this entity type between $c1$ and $c2$ and record the count as a property of the new *:DF_C* relationship.

```
1 MATCH (c1:Class) <-[:OBSERVED]- (e1:Event) -[df:DF]-> (e2:Event)
      -[:OBSERVED]-> (c2:Class)
2 MATCH (e1) -[:CORR]-> (n) <-[:CORR]- (e2)
3 WHERE n.EntityType = df.EntityType AND c1.Type = c2.Type
4 WITH n.EntityType as Type,c1,count(df) AS df_freq,c2
5 MERGE ( c1 ) -[rel2:DF_C {EntityType:Type}]-> ( c2 ) ON CREATE SET
      rel2.count=df_freq
```

Note that $c1$ and $c2$ can refer to the same node and thus self-loops are also included in the graph.

Observe that the aggregation query builds only on concepts of our data model, *Event*, *Class*, and *Entity* nodes and the relationships *DF*, *CORR*, *OBSERVED*, and requires no further domain knowledge of the underlying event data. In other words, the aggregation query demonstrates that the data model provides the right abstraction concepts for event data over multiple entities.

Because of this abstraction, the above query can be applied on "normal" entities as well as composite entities of reified relations, satisfying R14.

7.3 Demonstration on BPIC17

We demonstrate the aggregation queries on the BPIC17 dataset for which have already derived entities and *:DF* relationships for *Application, Workflow, Offer, Case_AO, Case_WO, Case_AW* (see Sect. 5.8), Resource, and the original case notion *Case_AWO* (see Sect. 6).

We consider two use cases: computing the handover-of-work social network between resources, and computing an "artifact-centric" directly-follows graph which distinguishes between different entity types.

Handover of Work We derived event *:Class*es for *Resource* as shown in Sect. 7.1. Note that the *:Class* nodes of type *Resource* are semantically different from the *:Entity* nodes of type *Case_R* created in Sect. 5.8, although we have one node of each per *e.resource* value.

We restricted the aggregation query of Sect. 7.2 to aggregate only the *:DF* relationships of *Case_AWO* for *:Class*es of type *Resource*. We thereby obtain the Handover-of-Work social network; the *count* property of the *:DF* relationship describes how often a resource handed work (of a specific *Case_AWO*) to another resource. The aggregation query had an execution time of 8.954 s.

We can retrieve this network with the query *MATCH (c1) -[dfc:DF_C]-> (c2) WHERE c1.Type = "Resource" AND c2.Type = "Resource" AND dfc.EntityType = "Case_AWO"*. We verified the correctness of the query using the social network mining plugin of ProM (www.promtools.org), see [52] for details. Figure 23 shows the Neo4j graph output of the query above on a sample of 20 cases.

With traditional event logs, creating a handover of work network typically requires the use of a tool or programming language, whereas Neo4j is capable of creating them by in-DB processing only.

Mining behavioral models over multiple entities In the same way, an aggregated directly-follows graph can be obtained in-DB by aggregating *:DF* relations. We aggregated the *:DF* relationships of Application, Workflow, Offer, Case_AO, Case_WO, Case_AW, Case_AWO for event class *Activity+Lifecycle*.

Figure 24(left) shows the classical directly-follows graph that can be obtained by aggregating all *:DF* relationships for the global case notion of the original log (entity type *Case_AWO*). Each node is a *:Class* node, and each edge is a *:DF_C* relationship of type *Case_AWO*; we only queried relationships with *count* $\geq$ 500. Figure 24(right) shows a process model discovered by the Split Miner [60] from the same event log; the model has a fitness of 95%, i.e., cannot explain 5% of the data.

However, both describe the behavior as a complex interleaving of steps of three different entities, while the underlying log suffers from convergence and divergence, see Sect. 2.2.

Figure 25 shows the graph we obtained by querying for the aggregated *:DF_C* relationships of types *Application* (dark

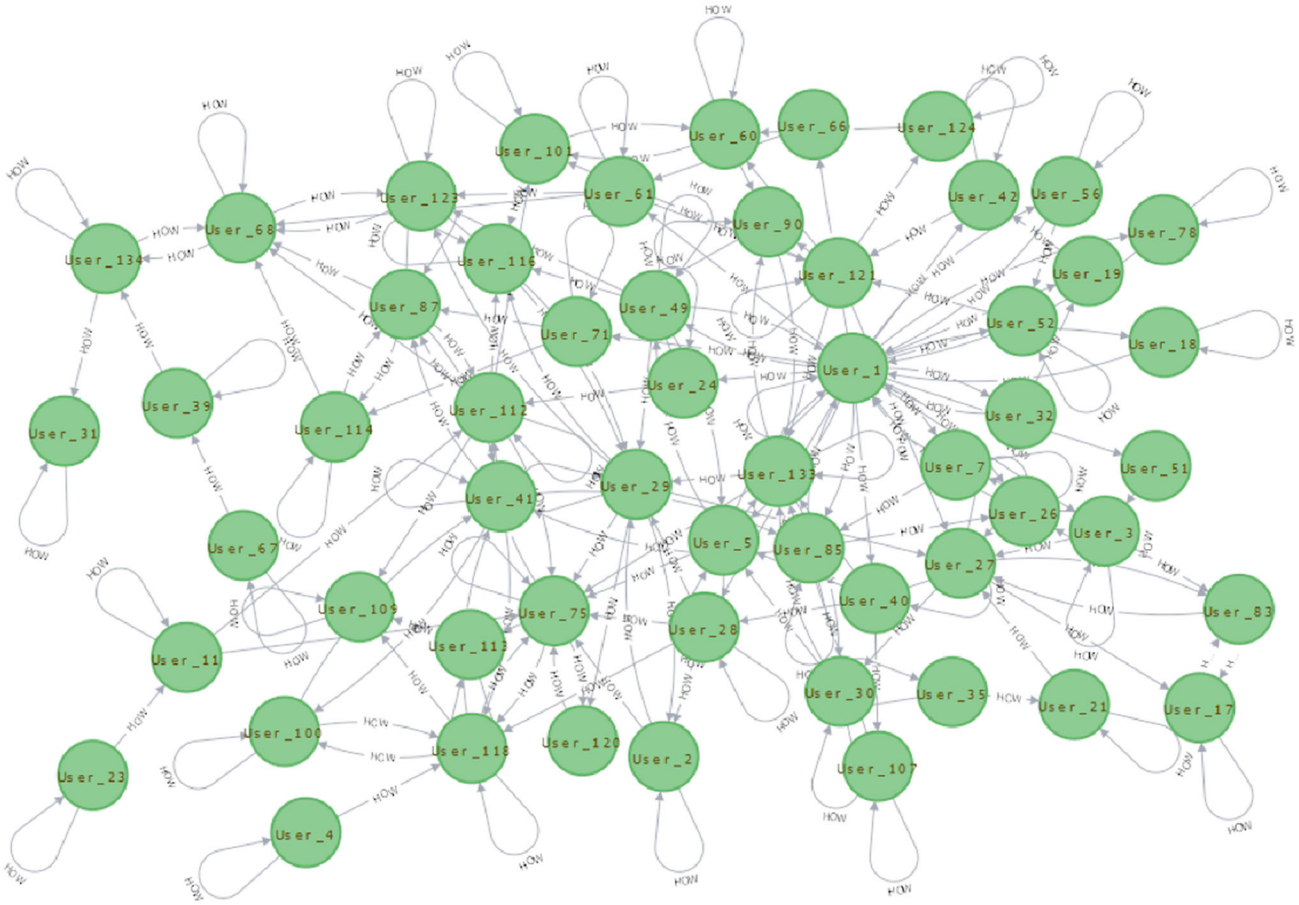


Fig. 23 Handover of work network

blue) *Workflow* (light blue), *Offer* (orange), and of the reified relations *Case_AO, Case_WO, Case_AW* (gray) with *count* $\geq$ 500, i.e., $>$ 98% of all process executions. This graph describes directly-follows relations per entity and thus is similar to an artifact-centric model [12] or a multiple viewpoint model [21].

Compared to Fig. 24, the graph of Fig. 25 explicitly describes the directly-follows behavior of each entity; the behavior of each entity is concurrent to the behavior of other entities up to the few explicit interactions shown by gray edges. In contrast, Fig. 24 shows few edges among the event classes of the same entity (Application, Workflow, or Offer) and most edges in between event classes of different entities because the classical event log interleaved all events. The graph of Fig. 25 is significantly easier to understand and more precise as it was derived from data without convergence and divergence.

8 Conclusion

We introduced a new data model for event data based on labeled property graphs. Our data model provides node types and relationship types (see Sect. 3) with semantic constraints (see Sect. 4) for all first-class concepts of event logs: events, entities (generalizing the case notion), event classes (generalizing the activity and the resource attribute), and the directly-follows relation between events related to the same entity only, satisfying requirements R0, R1, and R3 of Sect. 2.2. The semi-structured nature of graphs allowed us to represent multiple different, related entities (R2) and the relations between entities and events (R4) through dedicated correlation relationships. Thus, the data model can be seen as a *multi-dimensional event log* or *knowledge graph over events*, where events of each entity are ordered by "their" directly-follows relation leading to a *partial order* of events. Our data model avoids all shortcomings of existing event data models including event tables, event logs, and relational databases, see Sect. 2.3, while building on a standard data storage format. It specifically generalizes over XES event logs [2] in supporting multiple perspectives on the event data

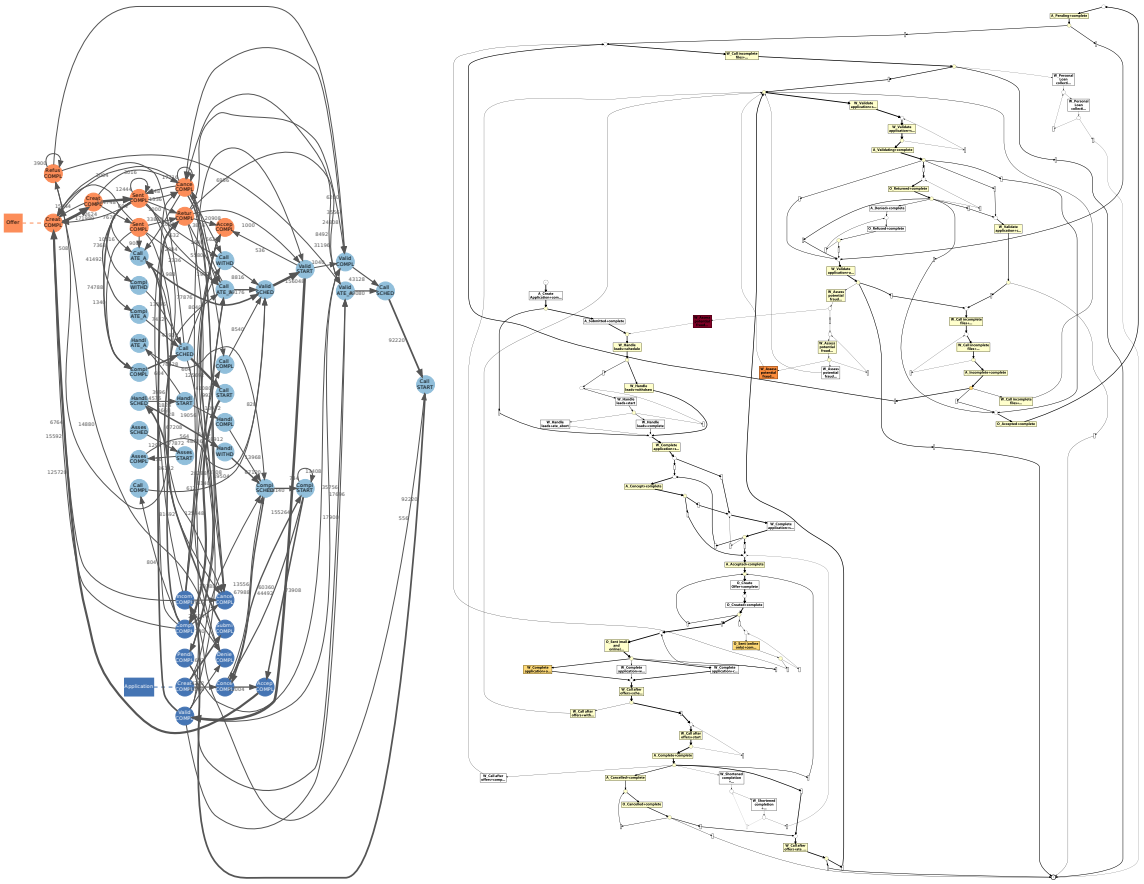


Fig. 24 Classical directly-follows graph (left) and process model discovered by Split Miner on classical event log of BPIC17 [28]

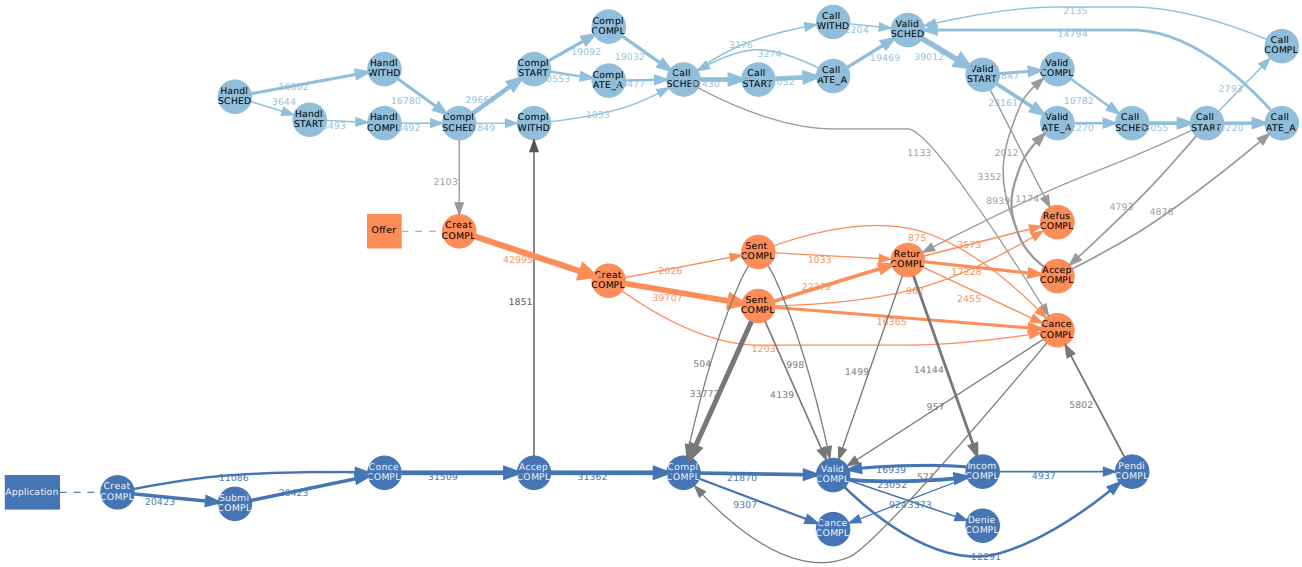


Fig. 25 Entity-centric directly-follows graph for Application, Workflow, Offer, and their interactions

from different case identifiers at once. The data model, however, has higher memory requirements and takes more time to load into a database than XES event log.

We provided a succinct set of queries to efficiently convert data in event table format into our data model (see Sect. 5. The queries are parameterized where user-provided domain-knowledge is required. We specifically provide queries

to reify relations between entities into composite entities allowing to derive directly-follows relations describing interactions between entities (R13). The data model and queries allowed us to convert represent 5 different real-life dataset into our data model.

We demonstrated that the query language Cypher allows querying event data in our data model, see Sect. 6. Queries and results are given as graphs, satisfying (R5). Queries Q4 and Q6 retrieve entire paths of events (R6) allowing to analyze the sequences. Q1–Q3 and Q6 select individual cases based on partial patterns (R7) allowing to "query by example." Q2, Q3, Q5, and Q6 query for temporal properties (R8) where Q5 specifically considers time; all queries correlate events related to a common entity (R9); Q7 queries aspects of multiple entities in the same query (R10) and allows to query behavior of multiple entities and combine results (R11). Altogether, we could demonstrate the queries over labeled property graphs satisfy R5–R11, which no existing query language on event data offers, see Sect. 2.3.

Finally, we demonstrated that our data model and Cypher allow aggregating events to event classes (R12) and directly-follows relations to event classes per entity (R14, R15). The resulting graphs are simpler and describe the behavior more accurately than techniques using other data models, see Sect. 7. *The queries and data sets are publicly available for further research*. [34–38,41].

The model has several limitations and requires further research. Our data model does not model properties of entities and semantics of relations between entities; practical applications require a more complete data model of this aspect as well. Within the scope of this work, we only consider converting event tables to our data model, whereas most event data is stored in relational databases; an automated technique for conversion is desirable for practical adoption. Furthermore, we currently completely rely on domain knowledge when loading the data into the graph database system. It is an open question how to leverage available context information or properties of the data to (partly) automate this process. Service-oriented platforms for event data such as ProcessAtlas [20] offer functionality that aid analysts in exploring the context information of the event data; a similar approach would be applicable for the proposed data model.

When one event is correlated to multiple entities of the same type, then the current modeling of the directly-follows relation does not distinguish between different individual entities. As a result, queries become more complex and aggregations of directly-follows relations lose information about these multiplicities; further research is required to aggregate behavior that preserves multiplicities of entities in interactions.

Cypher is highly expressive but not specifically designed for querying event data–it takes expertise and patience to write the right queries; query patterns and best practices have to be established. While we demonstrated feasibility and obtained performance that allows for usage in practice, existing graph database systems are still significantly slower than relational databases or dedicated algorithms, specifically due to deficiencies in query optimization which may easily render queries practically infeasible. Further improvements on the performance of graph databases are required, possible specifically taking the partially ordered nature of our data into account. Moreover, existing behavioral query languages such as PQL [40] offer dedicated constructs for behavioral querying that are not available as primitives in Cypher. It is an open question whether PQL's behavioral constructs can be expressed in Cypher (see Sect. 6.3) and which other higher-level primitives for querying multi-dimensional event data are desirable.

Our model also enables new lines of research. Providing a more general standard event data model allows for development of new event data analysis and process mining techniques that explicitly consider the presence of multiple entities. The data format enables the adoption of knowledge graph and graph mining techniques for event data.

Acknowledgements The results of this paper have been greatly influenced and shaped by discussion and inspirations over several years with Wil M.P. van der Aalst, Claudio di Ciccio, Marlon Dumas, Manuel Haug, Martin Klenk, Massimiliano de Leoni, Xixi Lu, Jan Mendling, Marco Montali, Alexander Rinke, and Stefan Schöning. The results would have been impossible without the careful preparation of the public BPI Challenge event data sets by Boudewijn van Dongen in preserving their multi-dimensional nature for this research, and without the regular availability of George Fletcher for introducing us to graph databases and providing insights into this field.

A False positives and false negatives in the directly-follows relation in BPIC17

We calculated the total number of directly-follows pairs (e_A, e_B) in the BPIC17 [28] data set for all events related to an offer (activity name starts with "O_"), denoted as "A df B for all events" in Table 6.

We calculated the total number of directly-follows pairs (e_A, e_B) in the BPIC17 data set for events related to an offer

Table 6 Share of globally wrong directly-follows pairs in sequential event log of BPIC17 (1) and share of missing directly-follows pairs in sequential event log of BPIC17 [28] (2)

A directly followed by B	O_Accepted	O_Cancelled	O_Create Offer	O_Created	O_Refused	O_Returned	O_Sent (mail and online)	O_Sent (online only)	Grand Total
1) Globally wrong DF edges (A df B for events of different offers / A df B for all events)									
O_Cancelled	**1/1 (100%)**	**4429/4429 (100%)**	**751/751 (100%)**			**5/5 (100%)**	**337/337 (100%)**	**11/11 (100%)**	**5534/5534 (100%)**
O_Create Offer				0/42995 (0%)					0/42995 (0%)
O_Created		353/754 (47%)	**3911/3911 (100%)**				44/36187 (0%)	1/1919 (0%)	4309/42771 (10%)
O_Refused					**975/975 (100%)**				**975/975 (100%)**
O_Returned	8/5227 (0%)	3/11 (27%)	**100/100 (100%)**			**65/65 (100%)**			176/5403 (3%)
O_Sent (mail and online)	**4/4 (100%)**	**335/384 (87%)**	**656/656 (100%)**			5/250 (2%)	**3111/3111 (100%)**	**1/1 (100%)**	**4112/4406 (93%)**
O_Sent (online only)	**8/8 (100%)**	**17/31 (55%)**	**151/151 (100%)**			4/270 (1%)	**1/1 (100%)**	**85/85 (100%)**	266/546 (49%)
Grand Total	21/5240 (0%)	**5137/5609 (92%)**	**5569/5569 (100%)**	0/42995 (0%)	**975/975 (100%)**	79/590 (13%)	3493/39636 (9%)	98/2016 (5%)	15372/102630 (15%)
2) Globally missed DF edges (A df B local per offer / A df B for all events)									
O_Cancelled	0/1 (0%)	0/4429 (0%)	0/751 (0%)	0/0 (oo)	0/0 (oo)	0/5 (0%)	0/337 (0%)	0/11 (0%)	0/5534 (0%)
O_Create Offer	0/0 (oo)	0/0 (oo)	0/0 (oo)	42995/42995 (100%)	0/0 (oo)	0/0 (oo)	0/0 (oo)	0/0 (oo)	42995/42995 (100%)
O_Created	0/0 (oo)	**1203/754 (160%)**	0/3911 (0%)	0/0 (oo)	59/0 (oo)	0/0 (oo)	**39707/36187 (110%)**	**2026/1919 (106%)**	**42995/42771 (101%)**
O_Refused	0/0 (oo)	0/0 (oo)	0/0 (oo)	0/0 (oo)	0/975 (0%)	0/0 (oo)	0/0 (oo)	0/0 (oo)	0/975 (0%)
O_Returned	**17228/5227 (330%)**	**2455/11 (22318%)**	0/100 (0%)	0/0 (oo)	**3573/0 (oo)**	0/65 (0%)	0/0 (oo)	0/0 (oo)	**23256/5403 (430%)**
O_Sent (mail and online)	0/4 (0%)	**16365/384 (4262%)**	0/656 (0%)	0/0 (oo)	**962/0 (oo)**	**22272/250 (8909%)**	0/3111 (0%)	0/1 (0%)	**39599/4406 (899%)**
O_Sent (online only)	0/8 (0%)	**875/31 (2823%)**	0/151 (0%)	0/0 (oo)	**101/0 (oo)**	**1033/270 (383%)**	0/1 (0%)	0/85 (0%)	**2009/546 (368%)**
Grand Total	**17228/5240 (329%)**	**20898/5609 (373%)**	0/5569 (0%)	42995/42995 (100%)	**4695/975 (482%)**	**23305/590 (3950%)**	39707/39636 (100%)	2026/2016 (100%)	**150854/102630 (147%)**

Results are reported as significant (bold) if the share/percentage is > 50%

(activity name starts with "O_") where e_A and e_B refer to *different* offers, denoted as "A df B for events of different offers" in Table 6. These are "false positive" directly-follows pairs as they do not describe a correct behavioral relation for an entity.

Choosing the offer id as case identifier and projecting the data to events related to an offer only, we calculated the total number of directly-follows pairs (e_A, e_B) per offer; denoted as "A df B local per offer" in Table 6. This is the ground truth of the directly-following events wrt. the offer entities.

Table 6(1) shows how many directly-follows pairs in the data ("A df B for all events") are *false positives* ("A df B for events of different offers"), i.e., directly-follows pairs in the data which do not exist in reality (as they relate events of different offers).

Table 6(2) shows the share of ground truth directly-follows pairs in the data ("A df B local per offer") over the directly-follows pairs in the sequential event log ("A df B for all events"). The discrepancy between both can be interpreted as *false negatives*, i.e., directly-follows pairs which exist but have not been reported in the data.

We observe several directly-follows pairs are reported always wrong, e.g., *O_Cancelled-O_Cancelled*, *O_Cancelled-O_Create Offer*, *O_Cancelled-O_Sent (mail and online)*, *O_Refused-O_Refused*, whereas other directly-follows pairs are always missing, e.g., *O_Returned-O_Refused*.

Overall, 15% of all directly-follows pairs are false positive, with some activities having a 100% false positive ratio. Further, the actual directly-follows relation has 47% more directly-follows pairs than represented in the sequential event log, where some activities show up to 39 times more directly-follows pairs than in the sequential event log (*O_Returned*).

References

1. van der Aalst WMP (2016) Process mining - Data Science in Action, 2nd edn. Springer, pp 3-452. ISBN 978-3-662-49850-7
2. Ieee standard for extensible event stream (xes) for achieving interoperability in event logs and event streams. IEEE Std 1849-2016 pp 1–50 (2016)
3. Bottrighi A, Canensi L, Leonardi G, Montani S, Terenziani P (2016) Trace retrieval for business process operational support. Expert Syst Appl 55:212–221
4. Deutch D, Milo T (2009) TOP-K projection queries for probabilistic business processes. In: ICDT 2009, ACM international conference proceeding series, vol 361, pp 239–251. ACM
5. Liu D, Pedrinaci C, Domingue J (2009) Semantic enabled complex event language for business process monitoring. In: 4th international workshop on semantic business process management, pp 31–34
6. Räim M, Ciccio CD, Maggi FM, Mecella M, Mendling J (2014) Log-based understanding of business processes through temporal logic query checking. In: OTM, LNCS, vol 8841, pp 75–92. Springer
7. Song L, Wang J, Wen L, Wang W, Tan S, Kong H (2011) Querying process models based on the temporal relations between tasks. In: EDOCW 2011, pp 213–222. IEEE Computer Society
8. Tang Y, Mackey I, Su J (2018) Querying workflow logs. Information 9(2):25
9. Augusto A, Conforti R, Dumas M, Rosa ML, Maggi FM, Marrella A, Mecella M, Soo A (2019) Automated discovery of process models from event logs: Review and benchmark. IEEE Trans Knowl Data Eng 31(4):686–705. https://doi.org/10.1109/TKDE.2018.2841877
10. Weerdt JD, Backer MD, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37(7):654–676. https://doi.org/10.1016/j.is.2012.02.004
11. Jans M, Soffer P (2017) From relational database to event log: Decisions with quality impact. In: BPM 2017 Workshops, LNBIP, vol 308, pp 588–599. Springer
12. Lu X, Nagelkerke M, van de Wiel D, Fahland D (2015) Discovering interacting artifacts from ERP systems. IEEE Trans Serv Comput 8(6):861–873
13. de Murillas EGL, Reijers HA, van der Aalst WMP (2016) Everything you always wanted to know about your process, but did not know how to ask. In: BPM Workshops, LNBIP, vol 281, pp 296–309
14. de Murillas EGL, Reijers HA, van der Aalst WMP (2019) Connecting databases with process mining: a meta model and toolset. Softw Syst Model 18(2):1209–1247
15. Dijkman RM, Gao J, Syamsiyah A, van Dongen BF, Grefen P, ter Hofstede AHM (2020) Enabling efficient process mining on large data sets: realizing an in-database process mining operator. Distrib Parallel Databases 38(1):227–253. https://doi.org/10.1007/s10619-019-07270-1
16. Schönig S, Rogge-Solti A, Cabanillas C, Jablonski S, Mendling J (2016) Efficient and customisable declarative process mining with SQL. In: Nurcan S, Soffer P, Bajec M, Eder J (eds) 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). https://doi.org/10.1007/978-3-319-39696-5_18
17. van der Aalst WMP (2019) Object-centric process mining: Dealing with divergence and convergence in event data. In: Ölveczky PC, Salaün G (eds) Software engineering and formal methods - 17th international conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11724, pp 3–25. Springer. https://doi.org/10.1007/978-3-030-30446-1_1
18. Li G, de Murillas EGL, de Carvalho RM, van der Aalst WMP (2018) Extracting object-centric event logs to support process mining on databases. In: Mendling J, Mouratidis H (eds) Information systems in the big data Era - CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, proceedings, lecture notes in business information processing, vol 317, pp 182–199. Springer (2018). https://doi.org/10.1007/978-3-319-92901-9_16
19. Popova V, Fahland D, Dumas M (2015) Artifact lifecycle discovery. Int J Cooperative Inf Syst 24(1):1550001:1–1550001:44. https://doi.org/10.1142/S021884301550001X
20. Beheshti A, Benatallah B, Motahari-Nezhad HR (2018) Processatlas: A scalable and extensible platform for business process analytics. Softw Pract Exp 48(4):842–866. https://doi.org/10.1002/spe.2558
21. Berti A, van der Aalst WMP (2020) Extracting multiple viewpoint models from relational databases. In: Ceravolo P, van Keulen M, López MTG (eds) Data-driven process discovery and analysis - 8th IFIP WG 2.6 international symposium, SIMPDA 2018, Seville, Spain, December 13-14, 2018, and 9th international symposium, SIMPDA 2019, Bled, Slovenia, September 8, 2019, Revised

selected papers, lecture notes in business information processing, vol 379, pp 24–51. Springer. https://doi.org/10.1007/978-3-030-46633-6_2
22. Esser S, Fahland D (2019) Storing and querying multi-dimensional process event logs using graph databases. In: Francescomarino CD, Dijkman RM, Zdun U (eds) Business process management workshops - BPM 2019 international workshops, Vienna, Austria, September 1-6, 2019, D, vol 362, pp 632–644. Springer. https://doi.org/10.1007/978-3-030-37453-2_51
23. Werner M, Gehrke N (2015) Multilevel process mining for financial audits. IEEE Trans Serv Comput 8(6):820–832. https://doi.org/10.1109/TSC.2015.2457907
24. Gonzalez Lopez de Murillas E (2019) Process mining on databases: extracting event data from real-life data sources. Ph.D. thesis, Department of Mathematics and Computer Science (2019). Proefschrift
25. Robinson I, Webber J, Eifrem E (2013) Graph databases. O'Reilly Media
26. van Dongen B (2014) BPI challenge 2014. Dataset. https://doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35
27. van Dongen B (2016) BPI challenge 2016. Dataset. https://doi.org/10.4121/uuid:360795c8-1dd6-4a5b-a443-185001076eab
28. van Dongen B (2017) BPI challenge 2017. Dataset. https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
29. van Dongen B (2018) BPI challenge 2018. Dataset. https://doi.org/10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972
30. van Dongen B (2019) BPI challenge 2019. Dataset. https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1
31. van Dongen B (2015) BPI challenge 2015. Dataset. https://doi.org/10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1
32. Cohen J, Dolan B, Dunlap M, Hellerstein JM, Welton C (2009) Mad skills: New analysis practices for big data. Proc VLDB Endow 2(2):1481–1492. https://doi.org/10.14778/1687553.1687576
33. Marín-Ortega PM, Dmitriyev V, Abilov M, Gómez JM (2014) Elta: New approach in designing business intelligence solutions in era of big data. Procedia technology 16:667 – 674. https://doi.org/10.1016/j.protcy.2014.10.015. http://www.sciencedirect.com/science/article/pii/S2212017314002424
34. Esser S, Fahland D (2014) Event graph of BPI challenge 2014. Dataset. https://doi.org/10.4121/14169494
35. Esser S, Fahland D (2015) Event graph of BPI challenge 2015. Dataset. https://doi.org/10.4121/14169569
36. Esser S, Fahland D (2016) Event graph of BPI challenge 2016. Dataset. https://doi.org/10.4121/14164220
37. Esser S, Fahland D (2017) Event graph of BPI challenge 2017. Dataset. https://doi.org/10.4121/14169584
38. Esser S, Fahland D (2019) Event graph of BPI challenge 2019. Dataset. https://doi.org/10.4121/14169614
39. Polyvyanyy A, Pika A, ter Hofstede AHM (2020) Scenario-based process querying for compliance, reuse, and standardization. Inf Syst 93:101563. https://doi.org/10.1016/j.is.2020.101563
40. Polyvyanyy A, ter Hofstede AHM, Rosa ML, Ouyang C, Pika A (2019) Process query language: design, implementation, and evaluation. CoRR arXiv:1909.09543
41. Esser S, Fahland D (2020) Event data and queries for multi-dimensional event data in the Neo4j graph database (Version 1.0). Dataset. https://doi.org/10.5281/zenodo.3865222
42. Fahland D (2019) Describing behavior of processes with many-to-many interactions. In: Donatelli S, Haar S (eds) Application and theory of petri nets and concurrency - 40th international conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, proceedings, lecture notes in computer science, vol 11522, pp 3–24. Springer (2019). https://doi.org/10.1007/978-3-030-21571-2_1
43. Syamsiyah A, van Dongen BF, van der Aalst WMP (2016) DB-XES: enabling process discovery in the large. In: Ceravolo P, Guetl C, Rinderle-Ma S (eds) Data-driven process discovery and analysis - 6th IFIP WG 2.6 international symposium, SIMPDA 2016, Graz, Austria, December 15-16, 2016, Revised selected papers, lecture notes in business information processing, vol 307, pp 53–77. Springer (2016). https://doi.org/10.1007/978-3-319-74161-1_4
44. Cuevas-Vicenttín V, Dey SC, Wang MLY, Song T, Ludäscher B (2012) Modeling and querying scientific workflow provenance in the D-OPM. In: 2012 SC Companion, pp 119–128. IEEE Computer Society
45. Huang X, Bao Z, Davidson SB, Milo T, Yuan X (2015) Answering regular path queries on workflow provenance. In: ICDE 2015, pp 375–386. IEEE Computer Society
46. de Murillas EGL, Hoogendoorn GE, Reijers HA (2017) Redo log process mining in real life: Data challenges & opportunities. In: Teniente E, Weidlich M (eds) Business process management workshops - BPM 2017 international workshops, Barcelona, Spain, September 10-11, 2017, Revised papers, lecture notes in business information processing, vol 308, pp 573–587. Springer. https://doi.org/10.1007/978-3-319-74030-0_45
47. zur Muehlen M (2009) Workflow management coalition - business process analytics format specification. Technical report, WfMC
48. Baquero AV, Molloy O (2012) Integration of event data from heterogeneous systems to support business process analysis. In: IC3K, CCIS, vol 415, pp 440–454. Springer
49. Beheshti S, Benatallah B, Motahari-Nezhad HR (2016) Scalable graph-based OLAP analytics over process execution data. Distrib Parallel Databases 34(3):379–423. https://doi.org/10.1007/s10619-014-7171-9
50. Beheshti S, Benatallah B, Nezhad HRM, Sakr S (2011) A query language for analyzing business processes execution. In: BPM 2011, LNCS, vol 6896, pp 281–297. Springer
51. Francis N, Green A, Guagliardo P, Libkin L, Lindaaker T, Marsault V, Plantikow S, Rydberg M, Selmer P, Taylor A (2018) Cypher: An evolving query language for property graphs. In: Management of data, pp 1433–1445. ACM
52. Esser S (2019) Using graph data structures for event logs. Capita selecta research project., Eindhoven University of Technology (2019). https://doi.org/10.5281/zenodo.3333831
53. van der Aalst WMP, Reijers HA, Song M (2005) Discovering social networks from event logs. Comput Support Coop Work 14(6):549–593. https://doi.org/10.1007/s10606-005-9005-9
54. van der Aalst WMP, Rubin VA, Verbeek HMW, van Dongen BF, Kindler E, Günther CW (2010) Process mining: a two-step approach to balance between underfitting and overfitting. Softw Syst Model 9(1):87–111. https://doi.org/10.1007/s10270-008-0106-z
55. Lu X, Fahland D, van der Aalst WMP (2014) Conformance checking based on partially ordered event data. In: Fournier F, Mendling J (eds) Business process management workshops - BPM 2014 international workshops, Eindhoven, The Netherlands, September 7-8, 2014, revised papers, lecture notes in business information processing, vol 202, pp 75–88. Springer (2014). https://doi.org/10.1007/978-3-319-15895-2_7
56. Pegoraro M, Uysal MS, van der Aalst WMP (2019) Discovering process models from uncertain event data. In: Francescomarino CD, Dijkman RM, Zdun U (eds) Business process management workshops - BPM 2019 international workshops, Vienna, Austria, September 1-6, 2019, revised selected papers, lecture notes in business information processing, vol 362, pp 238–249. Springer (2019). https://doi.org/10.1007/978-3-030-37453-2_20
57. Bonifati, A., Fletcher, G.H.L., Voigt, H., Yakovets, N.: Querying graphs. Synthesis lectures on data management. Morgan & Claypool Publishers (2018). https://doi.org/10.2200/S00873ED1V01Y201808DTM051
58. Angles R, Arenas M, Barceló P, Boncz PA, Fletcher GHL, Gutierrez C, Lindaaker T, Paradies M, Plantikow S, Sequeda JF, van Rest O, Voigt H (2018) G-CORE: A core for future graph query languages.

In: Das G, Jermaine CM, Bernstein PA (eds) Proceedings of the 2018 international conference on management of data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, pp 1421–1432. ACM. https://doi.org/10.1145/3183713.3190654

59. Polyvyanyy A, Weidlich M, Conforti R, Rosa ML, ter Hofstede AHM (2014) The 4c spectrum of fundamental behavioral relations for concurrent systems. In: Ciardo G, Kindler E (eds) Application and theory of petri nets and concurrency - 35th international conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings, lecture notes in computer science, vol 8489, pp 210–232. Springer. https://doi.org/10.1007/978-3-319-07734-5_12
60. Augusto A, Conforti R, Dumas M, Rosa ML, Polyvyanyy A (2019) Split miner: automated discovery of accurate and simple business process models from event logs. Knowl Inf Syst 59(2):251–284. https://doi.org/10.1007/s10115-018-1214-x