Storing and querying multi-dimensional process event logs using graph databases

Citation for published version (APA):

Document license:
CC BY

Document status and date:
E-pub ahead of print: 01/01/2019

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Storing and Querying Multi-Dimensional Process Event Logs using Graph Databases

Stefan Esser and Dirk Fahland
Eindhoven University of Technology, Eindhoven The Netherlands
s.esser@student.tue.nl, d.fahland@tue.nl

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 event sub-sequences 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 defined by temporal relations. In this paper, we report on an exploratory case study to store multi-dimensional event data in labeled property graphs and to query the graphs for structural and temporal properties together. Our main finding is that event data over multiple entities and identifiers with complex relationships can be stored in graph databases in a systematic way. Typical and advanced queries over such multi-dimensional event data can be formulated in the query language Cypher and can be executed efficiently, giving rise to several new research questions.

Keywords: process mining · event log · multi-dimensional processes · querying · graph databases

1 Introduction

Retrieving 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 activity, a case identifier referring to the object or case 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, the event data is single-dimensional and can be stored in an event log as a sequence of events. 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 [13, 19, 6, 4, 20, 17].

Most processes in practice however involve multiple inter-related entities which results in multi-dimensional event data in which each event is directly or indirectly linked to multiple different case identifiers; sequential event logs cannot represent such multi-dimensional event data [14]. Relational databases (RDBs) can store 1:n and n:m relations between events and case identifiers and among case identifiers — but the explicit behavioral information of sequences (of arbitrary length) is lost. Querying the full behavioral information from an RDB requires reconstructing the sequences through an arbitrary number of (self-)joins, can only be done under (severe) information
loss in the presence of multiple case identifiers [12, 14], and the required queries are large and non-intuitive [16]. Thus, querying and analyzing behavioral properties of multi-dimensional event data in an intuitive way is an open problem.

**State of the art.** A recent literature survey of 95 studies [10, Ch.7, pp.133][15] established requirements for querying event data. Focusing on querying for structure and behavior in multi-dimensional event data we derive from [10, pp.133] the requirements (R1) to query and analyze event data, and to (R2) consider relations between multiple data entities. The technique shall support (R3) storing and querying business process-oriented concepts (such as activities, cases, resources) and (R4) capture information about how events are related to different entities. 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 processes at the same time on the same dataset, and (R11) allow to query multiple event logs and combine results. Altogether, a user shall be able to query for individual events (and their properties), for different entities/case notions, for behavioral relations and patterns of multiple events (within and across entities).

Of the 95 works surveyed [10, pp.133], several approaches exist to retrieve cases from event logs (R7) for temporal properties (R8) [13, 19], for most frequent behavior [6], for sequences of activities [4] or algebraic expressions of sequence, choice, and parallelism over activities [20], or to check whether a temporal-logic property holds [17]. Several techniques support graph-based queries [5, 11, 13]. Yet, all these approaches only support a single fixed case notion and thus fail R2, R10, R11. Few techniques support querying over multiple entities or processes: the technique in [3] supports graph-based queries over event data from multiple entities, but does not allow to select individual cases or querying for behavioral properties (R7,R8). The language in [2] allows querying data from different processes, but cannot express properties of events or relations between events (R7, R8). DAPOQ [10, Ch.7] generalizes these approaches to query 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). No existing query language on sequential event logs or RDBs satisfies R1-R11.

**Hypotheses.** The above observations led us to the following hypotheses: (1) A graph-based event data storage format allows to explicitly represent both multi-dimensional relations between events and case identifiers and behavioral sequential information between events (as paths in the graph). (2) Queries can easily be formulated over such explicit representation of multi-dimensional event data.

**Method.** To test the validity of these hypotheses, we conducted an exploratory case study. We wanted to test the ability of labeled property graphs (LPGs), the data format of graph databases (GDBs) [18], to store structural and behavioral information together. And we wanted to test whether existing declarative query languages on labeled property graphs, such as Cypher [9] are expressive enough (see Sect. 2 for an introduction to GDBs and Cypher). Thus, we first formulated types of typical and advanced query operations over multi-dimensional event data that address R1-R11 in precise English
language (see Sect. 4). To test R1-R4, we had to represent an existing event dataset with multiple case identifiers with 1:1 and 1:n relations as an LPG; the dataset had to be representative of a real-life analysis. To test R5-R11, we had to define for each query type a corresponding query over LPGs and independently solved the same analysis questions through procedural programs. A query was valid when it led to the same result as the program. Further, we evaluated the size of the queries to assess how natural they allow solving each question.

**Results.** We selected the BPIC’17 [7] dataset of a loan application business process for this case study. It contains 3 entities Application (31,509 cases, 239,595 events), Workflow (31,509 cases, 768,823 events), Offer (42,995 cases, 193,840 events) each with their own identifier; Application is in a 1:1 relation with Workflow, and in a 1:n relation with Offer; further a common Case identifier subsumes all events of/related to the same Application into a single case. We chose the graph database Neo4j (neo4j.com) for LPG storage and querying due to off-the-shelf availability, Cypher support, and suitable performance. We iteratively developed mappings from the BPIC’17 event data to LPG concepts and formulated Cypher queries for all classes, until being able to answer all queries. In our mapping, each event and entity identifier in the BPIC’17 dataset becomes a different node in a LPG, and all structural relations (between entities), correlation (between entities and events), and temporal relations (between events) become labeled relationships in the LPG, satisfying R1-R4; see Sect. 3. We were able to answer each analysis question with a Cypher query that was valid compared to the procedural analysis. The query answer time always satisfied practical requirements on the full BPIC’17 dataset (answers within fractions of a second to at most a few seconds), and significantly outperformed our procedurally implemented single-pass search algorithms for more complex queries. Sect. 4 provides a summary of the queries and results; full details including implementation are available in a technical report [8]. We discuss limitations and alleys for future work in Sect. 5.

## 2 Process Event Logs and Graph Databases

**Event Logs.** Event logs consist of a sequence of events grouped into cases. Events are generated by information systems while executing a process. The transactions related to a specific process can be extracted as events of an event log of that process. Events have at least an activity attribute and a timestamp (or ordering attribute) describing which process step was executed and when (or in which order). Further attributes like resource and other event-specific data may be recorded as event attributes. A case in an event log represents a process instance with a unique identifier, e.g. order number or invoice number. Similar to events, cases can have a set of attributes that provide more information about the case instance such as the cumulative value of the invoice. Classical event logs enforce a single case identifier to which all events are related. Information systems usually host multiple uniquely identifiable entities, i.e., objects like orders and invoices. An entity or a combination of different entities can be used as case identifier. The creator of an event log is forced to choose under which perspective (case identifier) the process shall be analyzed, e.g. the invoice handling or the ordering aspects of the procurement process, and flattens the data accordingly [12]. The more different entities and potential
case identifiers are included, the more “remotely” related events are included, making the analysis of the flattened data harder and erroneous [14]. The BPI’17 log used in our case study (see Sect. 1) is flattened under the case identifier of an Application and events from all 3 entities are interleaved over time.

**Graph Databases.** *Labeled Property Graphs* (LPGs) are one of the data structures used in graph databases (GDBs) [18]. An LPG consists of nodes (vertices) and relationships (edges), each with an arbitrary number of key-value pairs, called properties.

We explain LPGs on the example of Fig. 1 which shows the relationships between a professor and 2 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. Figure 1 shows the LPG structure showing all components of the example. In the GDB system Neo4j used in our case study, a node of an LPG can have multiple labels while relationships have exactly one label and are directed.

**Querying Graph Databases.** Cypher is a language for querying LPGs [9] and supported by Neo4j. Cypher queries use pattern matching to select sub-graphs of interest. In the following, we explain the central Cypher query concepts used in the case study in a single (albeit inefficient) example query. Nodes are denoted in a query by round parentheses and can be used with variables, labels and properties like “(s: Student {Name: "Miro"})” where “s” is the variable name, “:Student” the node label and “{Name : "Miro"}” the key-value pair of the student’s Name property. Relationships must have a start and a destination node and may also be specified with variables, a name and properties. “(n1)−[:SUPERVISES]−>(n2)” for example represents the relationship node “n1” supervises “n2” with “n1” and “n2” being any 2 nodes in the graph. Cypher also allows to query the directed relationships in an undirected way: “(n1)−[:SUPERVISES]−(n2)”. For the example graph, we want define a query to return documents that Dirk co-authors with students, other than Stefan, working on these documents. We also want to retrieve the longest path that connects the matched students with Dirk. The query may be defined as follows:
The MATCH clause defines the pattern we want to retrieve from the graph. The pattern in line 1 includes students that have any type of relation to Dirk. The *-operator defines that the relationship between a student and Dirk may be direct or indirect over paths of arbitrary length and relationship types between Dirk and the student. 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 patterns. WITH in line 3 allows to process the matched patterns. In our example it only renames the variables, e.g. from “s”, to “student” to demonstrate that these variables serve as input for the subsequent clauses. This is how queries can be chained together, such that only Miro is handed over in the “student” variable, but Stefan is not. 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 is used to define the output of the nested query in lines 4-5. In the example graph, the student is Miro and the document is the paper. The paths variable 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. With this information we can sort the results by their path lengths in descending order with the ORDER BY clause and DESC option because the default ordering is ascending. Since we are interested in the longest path from Miro to Dirk only, we can limit the number of returned paths to 1 as shown in line 8. More detailed Cypher examples for the concepts used in the case study can be found in [8].

3 Representing (Multi-Dimensional) Event Data in a Graph DB

To represent the multi-dimensional event log data of BPIC’17 in an LPG, we developed the following mapping. We considered each entity identifier as a case identifier. Each event and each case (given by a case id) became a node in the LPG with a relationship “:EVENT_TO_CASE” to represent their correlation. Event nodes have, amongst others, timestamp and activity properties as they are event attributes in classical logs. According to the order in the sequential event log we define directly-follows relationships “:DF” between events correlated to the same case to describe the temporal order of the case. To represent the multi-dimensional nature of the data, we created for each case (defined by its own entity) its own “:DF” relation. For BPIC’17 this resulted in 4 different relations “:A_DF”, “:W_DF”, “:O_DF” (for the 3 entities) and “:DF” for the general case. This allowed us to query for behavior “along” each entity and across multiple entities. The “:DF” relations are enriched with a duration property to store the time between 2 events.

As BPIC’17 contains resources, a concept commonly used in organizational process mining [1, Sect. 9.3.1, pp.281], we added “Resource” nodes to enable querying for organizational dynamics such as handover of work. We therefore introduced handover of work relationships “:HOW” between resource nodes. As for “:DF”, we also defined
an entity-specific “:HOW” relation for each entity, Figure 2 shows the schema of the resulting LPG that we obtained. A more detailed description of the graph representation including all properties and relationships for this case study can be found in the technical report [8].

The above encoding represents events and top-level business process concepts (activity, case, resource) as nodes in a graph which we can query, satisfying (R1, R3); the semi-structured nature of graphs allows us to represent multiple different, related entities (R2), the relations between entities and events (R4), the correlation of events to the same entity (R9), and the correlation of multiple cases and entities to the same event (e.g., several events are correlated to both Case and Workflow). Thus, the graph database can be seen as a multi-dimensional event log, where events of each entity are ordered by “their” directly-follows relation leading to a partially ordered event log: the classical directly-follows relation connect events of different entities.

To import the BPIC’17 data into Neo4j, we used Cypher’s “LOAD CSV” clause which imports a given CSV file row by row such that clauses like “MATCH” or “WITH” can be used to filter or select the values and columns for creating nodes and relationships with the “CREATE” and “MERGE” clauses. “CREATE” creates all matching patterns extensively, including duplicates, and “MERGE” only creates patterns that do not exist already. “MERGE” creates patterns only once, e.g. for creating distinct case nodes. 2 approaches have been explored to create the graph. The first approach used the CSV load for several queries to create nodes and relationships by only loading the columns needed for the current query. This took roughly 75 minutes to complete on an Intel i7 CPU @ 2.8 GHz with 16 GB of memory. The second approach included saving additional columns as event node properties. This way we have been able to load data directly from the graph instead of using the CSV load. This approach significantly improved the cumulative execution time to 2:37 minutes. The additional properties were removed from the event nodes after the graph has been completely created and the execution of the respective query has already been accounted for in the 2:37 minutes. Please refer to [8] for a detailed description of the graph implementation and to https://github.com/multidimensional-process-mining/graphdb-eventlogs for the queries.

4 Querying (Multi-Dimensional) Event Data from a Graph DB

In the following paragraphs we present different classes of analysis questions designed to address requirements R5-R11 of Sect. 1 for querying (multi-dimensional) event data in LPGs. For each question we provide a Cypher query, report results and the query processing times (measured on an Intel i7 CPU @ 2.8 GHz machine with 16 GB of

**Q1. Query Attributes of Events/Cases.** We want to query an attribute of an individual case based on partial patterns to satisfy R7. By querying event and case attributes we make sure that the fundamental event log elements: case, event and their respective attributes are represented correctly in the graph data model and can be queried. The following query returns the event attribute “compleatetime” and the case attribute “loangoal” of Case “Application_681547497”.

```sql
MATCH (c: Case) <-[:EVENT_TO_CASE]- (e: Event)
WHERE c.name = "Application_681547497" AND e.activity = "ASubmitted"
RETURN e.compleatetime, c.loangoal
```

A case in BPIC’17 is a combination of the entities Application, Workflow and Offer and since we implemented every entity as distinct node type, we can also query attributes of a specific entity such as the offered amount of a given offer. The query has been processed in 0.083 seconds. After modifying the query to consider all cases, i.e. remove the condition for a specific case in line 2, the query completed in 0.944 seconds.

**Q2. Query Directly-Follows Relations.** Q2 is focused on temporal aspects. Here we want 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. Event x directly follows event y if there is no event in between them in the temporal order of the case. The next query returns the event directly following the node with the activity property “O_Facilitated” of a given offer entity by matching the :O_DF relationship.

```sql
MATCH (o: Offer) <-[:EVENT_TO_OFFER]- (e1: Event) <-[:O_DF]- (e2: Event)
WHERE o.name = "Offer_716078829" AND e1.activity = "O_Facilitated"
RETURN e1, e2
```

Directly-follows relations of other entities (Application and Workflow) or across entities (Case) can be queried by adjusting the query in the MATCH and WHERE clauses accordingly. The query execution time for one specific offer was 0.174 seconds whereas querying the “O_DF” relations with destination node “O_Facilitated” for all 42,995 offers took 11.291 seconds.

**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. Events x and y belong to the same case. If event y occurs after event x, then x is eventually followed by event y. In other words, if x and y are connected through a path of directly-follows relations of arbitrary length, they have an eventually-follows relation. We query the offer specific eventually-follows relationship between “O_Facilitated” and “O_Cancelling” for a given offer as follows:

```sql
MATCH (o: Offer) <-[:EVENT_TO_OFFER]- (e1: Event) <-[:O_DF*]- (e2: Event)
WHERE o.name = "Offer_716078829" AND e1.activity = "O_Facilitated" AND e2.activity = "O_Cancelling"
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 to match in the graph. For the given offer the query took 0.182 seconds. 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 4.469 seconds.

Q4. Case Variants. We want a query to return a case variant as path in the graph to satisfy R6. A variant is the sequence of its activities of a case. Case variants are for example used to detect frequent behaviour of a process. We can query the graph to retain the path of events of a case by walking over all of its “:`DF`” relationships from the first to the last event. For a given case this can be done as follows:

```
1 MATCH ( c : Case ) <[:EVENT_TO_CASE]−( e : Event ) <[:DF]−( e2 : Event)
2 WHERE NOT ( ) <[:DF]−( e ) AND NOT ( e2 ) <[:DF]−() AND c . name = 
   "Application_681547497"
3 RETURN ( e : Event ) <[:DF]−( 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. The query completed in 0.023 seconds. Similarly, we can query the graph for variants of another entity such as Offer. A query for all cases has not been tested since that would return the entire event log. There are simpler and more efficient ways to query for all events. Since variants typically do not come in the form of a path in a graph, we could for example further process the “paths” variable with list operators native to Cypher to turn the result in a list of activities of the nodes. UNWIND returns the individual items of the “paths” list and with a single path object the nodes() function can be used to return an ordered list of the nodes of a path. The list comprehension construct of Cypher can then be used to turn the list of nodes to a list of activities (or any other node property). This way we can generate a list object that can be compared for equality and used for other applications that case variants are typically used for.

Q5. Query Handover of Work. We want to show that, next to events, further event log concepts like resources can be correlated to the same process entity to satisfy R9. The handover of work social network is a technique of organizational process mining [1, Sect. 9.3.1, pp.281]. Resource nodes are used to create it. In fact the work social networks for one specific entity and across entities have already been created in form of the different “:`HOW`” relationships as described in section 3. By querying the graph for these “:`HOW`” relationships we get the respective social network. If we want to add the frequencies of handovers between 2 resources to the output, we can use a similar query to the one used to create the “:`HOW`” relationships as shown in [8]:

```
1 MATCH ( r1 : Resource ) <[:RESOURCE_TO_EVENT]−( e1 : Event ) <[:DF]−( e2 : Event)
   <[:RESOURCE_TO_EVENT]−( r2 : Resource )
2 RETURN ( r1 )−[:HOW]−( r2 ) as p, count(*) AS frequency
```

This query derives the handover of work network by aggregating “:`DF`” relationships such that we can count the “:`DF`” relationships between consecutive events and thus retrieve the number of handovers between their resources in the graph. The “:`HOW`” relationships only account for the information that there has been work handed over at
least once. Note that “r1” and “r2” can refer to the same node and thus self loops are also included in the network. With our graph model, it is also possible to define a query that derives the “.DF” relations from the “:HOW” relations. The query with frequency and path output, as shown above, had an execution time of 17.517 seconds.

A quicker version of the query only matching on “:HOW” relations without deriving the frequency of handovers took 1.066 seconds to complete. 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. Figure 3 shows the Neo4j graph output of the query above on a sample of 20 cases. Note that Neo4j by default visualizes all relationships between nodes in the result, even if they are not part of the returned subgraph, i.e. figure 3 shows also the entity specific handover of work relationships even though we explicitly specified “:HOW” relationships. This is only a matter of graphical representation, the data output of the query only contains the specified output format.

**Q6. 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 a process’ performance. For Q6 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:

```plaintext
MATCH (start:Event) -[:EVENT_TO_OFFER]-> (o:Offer) <-[:EVENT_TO_OFFER]-(end:Event)
WHERE start.activity = "O.Created" and end.activity = "O.Accepted"
3 WITH start, end, duration.between(start.completeTime, end.startTime) AS time, o
4 ORDER BY time DESC
5 LIMIT 1
6 RETURN start, end, time.days AS days, (toFloat(time.minutes)/60) AS hours, o
```

The query matches all “:EVENT_TO_OFFER” 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 [8]. The query for the elapsed time completed in 0.643 seconds. Querying for the longest path took 2.744 seconds.

**Q7. 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. For Q7 we want to satisfy R10 by combining the results from different (sub)logs and to satisfy R11 by querying 2 (sub)processes in a single query. We defined a query that returns all paths from “A_Create Application” to “O_Cancelled” of the BPIC’17 Cases for Offers that have “O_Created” directly followed by “O_Cancelled” on entity level, but only for those Cases that have more than one Offer with “O_Created” directly followed by “O_Cancelled”.

```
1 MATCH (e1: Event {activity: "O_Created"}) <[:DF]-(e2: Event {activity: "O_Cancelled"}) -[:EVENT_TO_OFFER]-> (o: Offer) -[:OFFER_TO_CASE]-> (c: Case)
2 WITH c AS c, count(o) AS ct
3 WHERE ct > 1
4 MATCH (: Event {activity: "O_Created"}) <[:DF]-(e: Event {activity: "O_Cancelled"}) -[:EVENT_TO_OFFER]-> (o: Offer) -[:OFFER_TO_CASE]-> (c)
5 WITH e AS O_Cancelled
6 MATCH p = (A_Create: Event {activity: "A_Create Application"})
7 RETURN p
```

The query also demonstrates how different queries can be combined. The first part returns all case IDs of Cases with more than one Offer and hands them over to second “MATCH” clause of the query. The case IDs are used to match all events with “O_Cancelled” activity of these Cases. The “O_Cancelled” events are the input for the last “MATCH” clause in line 6 which returns the paths from “A_Create Application” to the given “O_Cancelled” event nodes. This way we get a unique path for every Offer that meets the criteria. The query’s execution time was 0.453 seconds in Neo4j Browser. Figure 4 shows 4 of the 218 paths of the query’s output in Neo4j’s graphical representation. We want to point out that we have been able to replicate all query results with available process mining tools such as ProM or Disco, except for Q7. In order to replicate the query on multiple instance behavior we created a dedicated Python script which takes roughly 15 minutes to get the same results from the sequential log as the query above does in 0.453 seconds from the graph event data. Further details on the evaluation of Q1-Q7 can be found in the technical report [8]. With all above queries, we could demonstrate the ability of Cypher to express queries and results as graphs, satisfying (R5). Q4 and Q7 retrieve entire paths of events (R6) allowing to analyse the sequences. Q1-Q4 and Q7 select individual cases based on partial patterns (R7) allowing to “query by example”. Q2,Q3,Q6 and Q7 query for temporal properties (R8) where Q6 specifically considers time; all queries correlate events related to a common entity, Q5 shows that also process concepts such as resources...
can be used for correlation (R9); Q7 queries aspects of multiple processes in the same query (R10) and to query multiple logs and combine results (R11) by querying events of Applications and Offer events together.

We validated the correctness of our queries against an independent baseline implementation. The results of Q1-Q6 were obtained by processing the event log with manual filtering in Disco and social network mining algorithms in ProM. Q7 required a manual procedural algorithm using a single-pass search over the data as the evaluation with existing tools was not possible. In contrast, the graph analysis for Q1-Q7 required only Cypher queries with clauses and functions as described in [9] (except for a typecasts which are not part of Cypher but provided by Neo4j). Our Cypher queries obtained the same result as the baseline implementations; see [9] for details.

5 Discussion

We demonstrated the abilities of labeled property graphs and the query language Cypher to store and query multi-dimensional event data of multiple related entities and multiple case notions. We found that we can represent event data of multiple related entities and processes in a single graph database by encoding all first-level concepts of processes (entities, events, resources) as labeled nodes, and the established relations (directly-follows, executes, foreign-key relations between entities) as labeled relationships. This renders our mapping as a candidate for a multi-dimensional event log with a partial order between events of different entities. Further, we found that through the chosen relationships, Cypher allows querying paths and subgraphs of the event data based on patterns of nodes and relations. This allows to query cases of individual entities as well as paths across different, related entities, and even to construct aggregates such as the handover of work network. Compared to all prior works discussed in Sect. 1, encoding event data and process concepts alike in a graph database, and querying the graph database with the Cypher query language allowed us to address all requirements (R1-11). The latter suggests that graph databases are a suitable candidate for enabling multi-dimensional process mining on event data with multiple identifiers [14].

Limitations and future work. We obtained these results through carefully analyzing entities, events, and process concepts from a single event log using domain-knowledge about the studied event log [7]. While we believe the explored ideas to be generalizable, doing so raises a few research questions: How to systematically encode event data (related to multiple entities) into a graph database? How can a graph data schema for such a dataset be specified? A further limitation of this study is that we took a sequential event log as input, from which we reconstructed the relational structure. How to automatically translate event data from a relational database into a graph database in a systematic way is an open problem. Further, a broader understanding of analysis use cases on multi-dimensional event data and the required concepts for an useful and easy-to-use query language have to be established to generalize our findings beyond the concrete case and technology used.
References