Using graph data structures for event logs

Citation for published version (APA):

Document status and date:
Published: 20/05/2019

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.
Using Graph Data Structures for Event Logs

2IMI05 - Capita Selecta Analytics for Information Systems

Stefan Esser (1280163)
s.esser@student.tue.nl
# Table of contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Graph Data Structures for Event Logs</td>
<td>1</td>
</tr>
</tbody>
</table>

1 Introduction 1

2 Preliminaries 2
- 2.1 Event Log 2
- 2.2 Graph Database 2
- 2.3 Cypher 5
  - 2.3.1 Nodes and Relationships 5
  - 2.3.2 Clauses 5
- 2.4 Neo4j 9

3 Methods 10
- 3.1 Approach 10
- 3.2 Dataset 10
- 3.3 Insights 12

4 Representing Event Logs as Graphs 13
- 4.1 From Table to Graph 13
- 4.2 Data Preparation 16
- 4.3 Graph Creation 17

5 Querying Graph-based event logs using Cypher 19
- 5.1 Query Attributes of an Events 19
- 5.2 Query Attributes of Cases 20
- 5.3 Directly follows relations of events 21
- 5.4 Eventually follows relationships of events 21
- 5.5 Case variants 22
- 5.6 Handover of Work 23
- 5.7 Handover of Work based on Entities 24
- 5.8 Distance between two specific Activities 24
- 5.9 Involve multiple Entities 25

6 Conclusion 27
# Table of contents

## Using Graph Data Structures for Event Logs

### Appendix A  Data Preparation  
A.1 Data Conversion Problem  
A.2 Sample Set  
A.3 Instance Indices  
A.4 Python Script for Data Preparation  

### Appendix B  Implementation  
B.1 Sample Graph  
B.2 Full Graph  

### Appendix C  Screenshots  
C.1 Screenshots Questions 1 - 11  
C.2 Question 12  
C.2.1 Python Script  
C.2.2 Output  
C.2.3 Execution Times on Full Graph  

---

**TU/e**

Technische Universität Eindhoven  
University of Technology  

Where innovation starts
1 Introduction

Process mining as described in by Wil van der Aalst in [1] is a combination of data mining and business process management to a new discipline. The general purpose of process mining is to derive process insights from event data captured by information systems in the form of an event log. Three main applications of process mining are process discovery, process enhancement and process conformance checking. However, all techniques of these applications rely on event logs at some point. Event logs form the base of most techniques and are oftentimes created from multiple sources that are involved in the process that is subject to the research. Today, event logs usually consist of a table where every row represents a single event with activity name, case identifier, timestamp and optionally event attributes and case attributes. Event logs usually come in .csv or .xes format and reach from basic (activity, case id, timestamp) to extensive forms with different attributes for various activities and cases enriched with workflow lifecycle data. These event logs in table format can be queried with SQL, but interesting queries in the process context like "cases where activity A is followed by activity B" cannot be formulated. This form of storing data is not considered optimal since it does not implement the second normal form used in database normalization, nor does it generally allow for direct application of basic process mining techniques like the creation of a directly follows graph of activities while retaining the specific event data. This report explores graph data structures as alternative format to store event logs as base for process mining. These graph-based event logs are henceforth referred to as graph event logs.

The main contribution of this report is to address and answer the following research questions:

1. How can the concepts of event logs be represented in graph databases?
2. How to use graph-based queries to answer process analysis questions within and across cases?

https://www.overleaf.com/project/5be95b119f0dd86be9c4bf4 Research question 1 can be broken down into two sub-questions:

1.a How can cases and the temporal and causal ordering of events be represented in graphs?
1.b How can attributes and values be stored and shared across cases?

In chapter 2 we provide preliminary information about event logs for process mining and property graph data structures. Chapter 3 introduces the case study, i.e. BPI Challenge 2017 (BPIC17)\(^1\) dataset, and the methods used for this research assignment. The transition from the sequential event log to a graph event log based on the case study is described in chapter 4. Chapter 5 is dedicated to querying a graph event log to evaluate our approach. In chapter 6 we draw the conclusions of this research assignment and give an outlook on how the graph-based representation of event data and queries could be used in the future of process mining.

\(^1\) https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
2 Preliminaries

2.1 Event Log

The notion of event logs used for this report is based on [1]. In event logs for process mining we assume that a log contains the events of a single process with case identifiers and activities as minimum required elements of the log such that the combination of activity and case represent an event. In other words, a case in the event log represents a process instance and an activity refers to a task within a process instance. Table 2.1 shows a basic example of an event log. Every row represents an event in an application process for loans. A_Create Application is the first activity of case 430577010, which consists of 7 events in total. The "time" column adds information about the timing of the events. In the example in table 2.1 we have the timestamp on task completion, other event logs may contain additional timestamps e.g. for the start of a task allowing to generate more insights on the respective activities like throughput times. However, timestamps don’t belong to the set of minimum required elements since we generally assume that the events are grouped by case and the order of activities in the log corresponds to the temporal order in which they have been observed in the system. The sequence of activities within a case is called a trace. The example log has 2 trace variants, because cases 430577010 and 1639247005 have an identical sequence of activities while the sequence of activities in case 180427873 is different. A log can contain any number of columns. In our example log we have 3 more columns, "resource", "amount" and "offer". These columns are called attributes. In general we differentiate between event attributes and case attributes. The "resource" column represents the user that generated the event and probably also executed the respective activity which is a typical event attribute that can be found in many event logs. The "amount" column provides the loan value that has been applied for in the loan application and thus represents a case attribute. The information that this is a case attribute is not explicitly contained in the .csv table and has been derived from the fact that this value does not change throughout all events of a case in the log. Not every event needs to have the same set of attributes. This becomes clear when we inspect the "offer" column. It belongs to the event attribute category, since O_Create Offer of case 430577010 has the offer id 104557954 and the offer id is only used for certain activities. There can also be more than one offer like in case 180427873, indicating that an offer should rather be treated as a separate object or entity of the process which might also have its own attributes contained in the log. In sequential event logs information about such process specific entities and their affiliated attributes is only contained implicitly. We use the notion of an entity to describe process related objects that have no predefined standard notion in event logs. An entity of a process can be any object or concept with distinct existence related to the process, e.g. an invoice, an offer or a workflow.

Event logs can, for example, be used to discover process models as described in chapters 6 and 7 of [1] or compared with a reference model to determine whether log and model fit as profoundly discussed in [2]. Depending on how much information is contained in the event log further information on the process can be derived, such as hand-over of work relationships of resources or the general process performance. Please refer to chapter 9 of [1] for further information on these process mining perspectives.

2.2 Graph Database

Generally, a graph is a collection of vertices and edges. In the context of graph databases (DB), vertices are often referred to as nodes and edges as relationships. Graphs can be represented in a database in different ways like hypergraphs, property graphs or triples. The database management system (DBMS) we used for this research assignment makes use of a specific type of property graph, thus we introduce the concept of graph databases with the help of a labeled property graph, which has the following characteristics [3]:

2
Table 2.1: Event Log Example

<table>
<thead>
<tr>
<th>case</th>
<th>activity</th>
<th>time</th>
<th>resource</th>
<th>amount</th>
<th>offer</th>
</tr>
</thead>
<tbody>
<tr>
<td>430577010</td>
<td>A_Create Application</td>
<td>17.03.2016 17:32:13</td>
<td>User_24</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>430577010</td>
<td>O_Create Offer</td>
<td>17.03.2016 19:19:42</td>
<td>User_24</td>
<td>5000</td>
<td>104557954</td>
</tr>
<tr>
<td>430577010</td>
<td>O_Created</td>
<td>17.03.2016 19:19:43</td>
<td>User_24</td>
<td>5000</td>
<td>104557954</td>
</tr>
<tr>
<td>430577010</td>
<td>O_Sent (mail and online)</td>
<td>17.03.2016 19:21:56</td>
<td>User_24</td>
<td>5000</td>
<td>104557954</td>
</tr>
<tr>
<td>430577010</td>
<td>A_Incomplete</td>
<td>23.03.2016 17:02:03</td>
<td>User_101</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>430577010</td>
<td>O_Accepted</td>
<td>24.03.2016 10:11:02</td>
<td>User_116</td>
<td>5000</td>
<td>104557954</td>
</tr>
<tr>
<td>430577010</td>
<td>A_Pending</td>
<td>24.03.2016 10:11:02</td>
<td>User_116</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>1639247005</td>
<td>A_Create Application</td>
<td>22.04.2016 12:40:22</td>
<td>User_61</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>1639247005</td>
<td>O_Create Offer</td>
<td>22.04.2016 16:40:37</td>
<td>User_61</td>
<td>10000</td>
<td>110013904</td>
</tr>
<tr>
<td>1639247005</td>
<td>O_Created</td>
<td>22.04.2016 16:40:38</td>
<td>User_61</td>
<td>10000</td>
<td>110013904</td>
</tr>
<tr>
<td>1639247005</td>
<td>O_Sent (mail and online)</td>
<td>22.04.2016 16:41:30</td>
<td>User_61</td>
<td>10000</td>
<td>110013904</td>
</tr>
<tr>
<td>1639247005</td>
<td>A_Incomplete</td>
<td>23.04.2016 10:33:30</td>
<td>User_61</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>1639247005</td>
<td>O_Accepted</td>
<td>29.04.2016 16:08:15</td>
<td>User_90</td>
<td>10000</td>
<td>110013904</td>
</tr>
<tr>
<td>1639247005</td>
<td>A_Pending</td>
<td>29.04.2016 16:08:15</td>
<td>User_90</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>180427873</td>
<td>A_Create Application</td>
<td>06.09.2016 10:21:43</td>
<td>User_56</td>
<td>26000</td>
<td></td>
</tr>
<tr>
<td>180427873</td>
<td>O_Create Offer</td>
<td>06.09.2016 11:29:39</td>
<td>User_56</td>
<td>26000</td>
<td>1156783066</td>
</tr>
<tr>
<td>180427873</td>
<td>O_Created</td>
<td>06.09.2016 11:29:39</td>
<td>User_56</td>
<td>26000</td>
<td>1156783066</td>
</tr>
<tr>
<td>180427873</td>
<td>O_Create Offer</td>
<td>06.09.2016 11:45:21</td>
<td>User_56</td>
<td>26000</td>
<td>1156783067</td>
</tr>
<tr>
<td>180427873</td>
<td>O_Created</td>
<td>06.09.2016 11:45:21</td>
<td>User_56</td>
<td>26000</td>
<td>1156783067</td>
</tr>
<tr>
<td>180427873</td>
<td>A_Cancelled</td>
<td>07.10.2016 8:00:17</td>
<td>User_1</td>
<td>26000</td>
<td></td>
</tr>
<tr>
<td>180427873</td>
<td>O_Cancelled</td>
<td>07.10.2016 8:00:17</td>
<td>User_1</td>
<td>26000</td>
<td>1156783066</td>
</tr>
<tr>
<td>180427873</td>
<td>O_Cancelled</td>
<td>07.10.2016 8:00:17</td>
<td>User_1</td>
<td>26000</td>
<td>1156783067</td>
</tr>
</tbody>
</table>

- A graph consists of nodes and relationships
- Each node and relationship can have an arbitrary number of properties in the form of key-value pairs
- A node can have one or multiple labels
- A relationship always has a start and end node and thus is directed
- Every relationship has a name

We want to illustrate these characteristics with the help of relationships between a paper and its authors shown in figure 2.1. The circles are the nodes and the arrows are the edges. The example contains nodes with different labels, i.e. :Person, :Professor, :Student and :Document. The labels enable us to group nodes with the same label. The coloring in figure 2.1 is to illustrate that Dirk and Stefan belong to the group of nodes with the label :Person and the Paper has no label in common with other nodes. By referring to nodes as Dirk or Stefan we already introduced the next characteristic in our example, namely properties. In the example all nodes with the :Person label have a property called name and the :Document has a type. However, the visualization of graph structures often require some abstraction in order to keep it readable, because the properties of the nodes are not limited to what is shown. The Paper node, for example, could have other properties like title or abstract and a :Person node could have an address property. The same applies to the properties of relationships, since Dirk :SUPERVISES Stefan as of 02.09.2018 and the relationship might get a property end_date as soon as this report is finished. Nodes with the same label do not necessarily need to have the same set of properties, but from a data modeling perspective it is useful ensure some form of consistency such that nodes with the same label also have the same attributes defined.
While figure 2.1 illustrates a specific set of nodes and relationships, it does not give us the complete picture of the data structure. Including every detail would heavily reduce the readability when the set of labels, nodes, properties and relationships grows. For this representation, we opt for a table format. This requires one table for the nodes and one for the relationships. For our example this might look as follows:

Table 2.2: Example Graph Nodes Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Person</td>
<td>Name*</td>
</tr>
<tr>
<td></td>
<td>Address</td>
</tr>
<tr>
<td>:Professor</td>
<td>EmployeeID*</td>
</tr>
<tr>
<td></td>
<td>Department</td>
</tr>
<tr>
<td></td>
<td>Group</td>
</tr>
<tr>
<td>:Student</td>
<td>StudentID*</td>
</tr>
<tr>
<td></td>
<td>Program</td>
</tr>
<tr>
<td>:Document</td>
<td>Title*</td>
</tr>
<tr>
<td></td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
</tr>
</tbody>
</table>

Note that the properties marked with an asterisk* in table 2.2 might serve as unique identifier for a node. However, when storing this structure in a DBMS there might also be some internal unique identifier, hence a specification of a unique id is not required.

Relationships need a source and a destination node. Name and direction of a relationship corresponds to the type of a relationship. Generally, the type does not limit a relationship to originate from or end at nodes with predefined labels. However, to develop a meaningful schema for the graph database we want to specify between which type of nodes (labels) what type of relationship (name) can be established. Table 2.3 shows the relations of our example.

As we can see in table 2.3, relationships are not required to have properties. The same applies to nodes. With figure 2.1, table 2.2 and table 2.3 we showed exemplary how a labeled property graph structure can be represented. The fact that relations between individual data points and the data points themselves are equally important in graph databases allows to create many different views on the same data in a more flexible way than we are used to from relational databases.
Using Graph Data Structures for Event Logs

<table>
<thead>
<tr>
<th>Start Node Label</th>
<th>Name</th>
<th>End Node Label</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Professor</td>
<td>:SUPERVISES</td>
<td>:Student</td>
<td>start_date, end_date</td>
</tr>
<tr>
<td>:Person</td>
<td>:IS_AUTHOR_OF</td>
<td>:Document</td>
<td></td>
</tr>
<tr>
<td>:Person</td>
<td>:IS_COAUTHOR_OF</td>
<td>:Document</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Example Graph Relationships Table

2.3 Cypher

"Cypher is a declarative query language for property graphs. Cypher provides capabilities for both querying and modifying data, as well as specifying schema definitions." [4]. In the following paragraphs we introduce all relevant aspects of Cypher w.r.t. this paper.

2.3.1 Nodes and Relationships

Cypher recognizes nodes as round brackets "()". Where the empty node "()" represents a wildcard for any node label. To narrow the selection of nodes down, Cypher provides different options. "(:Person)" limits the results to the nodes with the label :Person and "(:Person {Name: 'Stefan'})" takes a property into account and will take only the nodes with label :Person and name Stefan. Selection of nodes based on properties without specifying a label is also possible. The notion of nodes may take multiple labels as well as multiple properties into account that can be generally represented as "(:Label_1, :Label_2, ..., :Label_n {Property_1: '#Value_1', Property_2: '#Value_2', ..., Property_n: '#Value_n'})". If we need the resulting nodes for later clauses, we can add a variable name before the first colon "(p:Person {Name: 'Stefan'})". "p" could then be used to refer to all nodes with label :Person and name Stefan. If we want to assign all nodes to a variable, regardless of their properties or labels, we can use the node wildcard "()" with a variable name "n" to get "(n)".

Relationships are represented by dashes. "−−" is the undirected wildcard and "<−>" or "−→" are directed wildcards. As mentioned earlier, a relationship requires a start and an end node in a labeled property graph. Thus, the minimum required elements to represent a relationship are two nodes and one relationship "(−−)". This pattern matches all relationships without any restrictions in their properties, labels, directions or start and end nodes. Similarly to nodes, we can make the selection criteria for relationships more strict in terms of direction "(−→)", relationship name "(:SUPERVISES)->()", properties "()-[start_date: '02-09-2018']-()" or properties and relationship name "()-[:SUPERVISES {start_date: '02-09-2018'}]-()". Note that for relationships only one name exists, which defines the type of the relationship and their number of properties can be zero or many. Relationships can also be stored in variables in a similar way like nodes: "()-[r:SUPERVISES]-()" and assigning a whole path to a variable can for example be done by "p = (:Person)−→(:Person)-[:IS_AUTHOR_OF]-(:Document)".

2.3.2 Clauses

Cypher implements a large set of clauses. This section introduces the subset of clauses that have been used for this research based on the Neo4j Cypher Manual [5] and on the paper by Francis et al. [4].

MATCH

"MATCH" specifies the patterns to be searched for in the database. It is the matching command that has implicitly been used to introduce the node and relationship representation in section 2.3.1. The "MATCH" clause can be aliased with the "AS".
MATCH (p:Professor) -[:SUPERVISES]->(:Person {Name: 'Stefan'}) AS SupervisorsToStefans

This line will match any node that has a :SUPERVISES relationship to a :Person with the name Stefan. The resulting set of patterns can be referred to as "SupervisorsToStefans" and the matched :Professor nodes can be referred to as "(p)" in consecutive clauses. However, the "MATCH" clause alone will not return or manipulate any data. It only matches patterns.

RETURN

"MATCH" followed by a "RETURN" clause is one of the most basic queries in Cypher. "RETURN" receives the set of patterns matched by "MATCH".

MATCH (p:Professor) -[:SUPERVISES]->(:Person {Name: 'Stefan'})
RETURN (p)

The query above will return all :Professor nodes as variable "(p)". This result may contain duplicates. In our example, Dirk would be included twice if he supervised a second :Student with name Stefan. If this is not desired, "DISTINCT" can be added to the "RETURN" clause.

MATCH (p:Professor) -[:SUPERVISES]->(:Person {Name: 'Stefan'})
RETURN DISTINCT (p)

Now Dirk will only be returned once, regardless of the number of matched patterns Dirk is included.

WITH

The "WITH" clause can manipulate the results from its preceding clause before it hands it to the succeeding clause. Thus, "WITH" is always located between two other clauses.

MATCH (p:Professor) -[:SUPERVISES]->(:Person {Name: 'Stefan'})
WITH COUNT(r) AS NoOfSupervisions, (p)
RETURN (p), NoOfSupervisions

The COUNT() function aggregates the matched "-[r:SUPERVISES]->" relationships per "(p)" providing the count of the aggregated relationships. Cypher offers an extensive set of predefined functions and also allows the definition of user-defined queries. Please note that the example query above serves the purpose of illustrating the use of "WITH". The exact same result can also be produced by using the following query:

MATCH (p:Professor) -[:SUPERVISES]->(:Person {Name: 'Stefan'})
RETURN (p), COUNT(r) AS NoOfSupervisions

Both queries count the number of :SUPERVISES relationships with respect to (p) and set the alias "NoOfSupervisions" to the result. The difference between the two queries is the data handed over to the "RETURN" statement. While the latter query can make use of the "r" variable, i.e. the individual :SUPERVISES relationships in the patterns delivered by the "MATCH" clause, the first query's "WHERE" only has the aggregated count available.

WHERE

"WHERE" is a sub-clause of reading clauses ("MATCH" and "WITH"). It is used to put constraints to the patterns matched by "MATCH" or to filter the results of a "WITH" clause. Note that the effect of "WHERE" depends on the preceding reading clause. "WHERE" following "MATCH" constrains every single pattern while matching, i.e. only patterns that conform to the "WHERE" are matched. When
using "WHERE" after "WITH", the "WHERE" clause filters from the whole set of patterns, i.e. the patterns get matched according to the "MATCH" clause and then filtered by "WHERE".

```
MATCH (p:Professor)-[r:SUPERVISES]->(:Person {Name: 'Stefan'})
WITH COUNT(r) AS NoOfSupervisions, (p)
WHERE NoOfSupervisions > 1
RETURN (p)
```

Suppose the data structure of our example in section 2.1 does not allow for more than one :SUPERVISES relationship between a :Student and a :Professor. Then, the statement above will return all :Professor nodes that supervise more than one different Stefan. If the data structure would allow multiple :SUPERVISES relationships between two individual nodes, say for every supervised :Document one to enrich them with document specific properties, then the statement would return all :Professor nodes that supervise the creation of at least two documents of any Stefan, which is possibly but not necessarily the same person.

**ORDER BY**

"ORDER BY" is another sub-clause. It can be used following "RETURN" or "WITH" and, as its name suggests, orders the output of the respective clause.

```
MATCH (p:Professor)-[r:SUPERVISES]->(:Person {Name: 'Stefan'})
RETURN (p), COUNT(r) AS NoOfSupervisions
ORDER BY NoOfSupervisions DESC
```

By default the output is in ascending order, by adding "DESC[ENDING]" after the variable to sort on the order can be reversed. The above query will return the professors supervising a Stefan and the respective number of supervisions in descending order.

**LIMIT**

"LIMIT" constrains the number of output rows.

```
MATCH (p:Professor)-[r:SUPERVISES]->(:Person {Name: 'Stefan'})
RETURN (p), COUNT(r) AS NoOfSupervisions
ORDER BY NoOfSupervisions DESC
LIMIT 1
```

The above query limits the output to one. The output will be one of the professors with the most supervisions in the matched patterns.

**CREATE**

With "CREATE" nodes, relationships and complete paths can be created by using their respective representations introduced in section 2.3.1.

The following statement creates a single node with two labels and the name Miro:

```
CREATE (:Student:Person {Name: 'Miro'})
```

The newly created node Miro is is yet disconnected from the other nodes as shown in figure 2.2.
The following statement exemplifies the creation of the relationship between Dirk and Miro when both nodes already exist in the graph, i.e. only the relationship is created. Resulting in the graph illustrated in figure 2.3.

```
MATCH (p:Professor {Name: 'Dirk'}), (s:Student:Person {Name: 'Miro'})
CREATE (p)−[:SUPERVISES]−>(s)
```

If we desire to create a node and a relationship between an existing and that new node in one go, we can use a different approach. The next statement directly creates a :SUPERVISES relationship between the existing :Professor node named Dirk and the new node Miro. Note that this query will create a new node for Miro, regardless if a similar node already exists because Miro has not been included from the "MATCH" clause.

```
MATCH (p:Professor {Name: 'Dirk'})
CREATE (p)−[:SUPERVISES]→(:Student:Person {Name: 'Miro'})
```

Suppose this query has been performed on the graph in figure 2.1. The resulting graph will have the structure shown in figure 2.3.
MERGE

Simply said, "MERGE" can be seen as a combination of "MATCH" and "CREATE". It ensures the existence of the pattern specified in the clause. Consider the graph shown in figure 2.2 as starting point and we want to have a :SUPERVISES relationship between Dirk and all students in our graph, but only if such a relationship does not exist already.

```plaintext
1 MATCH (d:Professor:Person {Name: "Dirk"}), (s:Student)
2 MERGE (d)-[:SUPERVISES]->(s)
```

Because the specified pattern in the "MERGE" clause already exists for Dirk and Stefan, no new relationship is created for that match. Only Dirk and Miro are connected with a new relationship resulting in the graph shown in figure 2.3.

LOAD CSV

"LOAD CSV" is used to import csv files. The clause can be used with various parameters to cope with the different variants of csv files. The "LOAD CSV" clause needs a variable specified with "AS", similar to the aliases introduced earlier. Similar to the "MATCH" clause with a variable, the aliased variable contains the set of matched patterns which are all rows from the csv file in this case. To work with specific columns one can specify this by column index "variable[index]" or, if the csv contains column headers, by column name "variable[columnName]".

2.4 Neo4j

Neo4j is the graph DBMS used for this research. Neo4j, Inc. describes its product as an ACID compliant DBMS with highly performant read and write scalability that relies on a native graph storage [6]. Cypher has its roots at Neo4j [4]. Neo4j Desktop is suitable to install a local database and comes with Neo4j Browser, a console tool that enables the user to run queries on a database and inspect results with a basic visualization.
3 Methods

In this chapter we present our approach to explore how to represent and access event log information in a property graph. We combine the concepts of event logs and property graphs introduced in section 2 to learn how, for example, the temporal ordering of events or their affiliation with a specific case can be represented as a set of nodes and relationships with properties and labels.

In section 3.1 we provide an overview of our approach. Section 3.2 describes the data set we used as case study for this research and in section 3.3 we define a set of insights we want to get from the data of our case study. These insights are aligned to the event log concepts and form the base for the graph data model for the log.

3.1 Approach

We used the BPIC17 as case study to conduct our research. The log contains, next to the events directly related to the main process instance, additional workflow information and events that are related to a sub process for offers. The aim of this research is to explore the feasibility of using graph databases to store event logs for process mining. Property graphs allow to flexibly model your data and thus design decisions need to be taken, e.g. a column in an event log table might become a property of a node type or a node type itself. Yet, no schema or guideline exists how to transform a sequential event log into a property graph representation. Therefore, we base our design decisions on the information we want to retrieve from the respective case study data set. Thus we first define a set of typical insights we want to get from the log and create the graph model based on these insights. To create the graph based log and access the information from the graph we use Cypher as query language and resort to the query concepts introduced in 2.3.2. Besides some data preparation steps, such as adding indices for sequences to the events of identified process entities to the event log table, we solely use Cypher queries to create the graph and access its information. We use Neo4j as DBMS for the implementation.

To verify that the information of the log is still contained in the graph event log, a fixed subset of 20 cases of the BPIC17 will be transferred to the new graph structure and queried with Cypher to generate the previously defined insights and the results are compared with results from traditional process mining tools like ProM or Disco used with the original event log.

3.2 Dataset

BPIC17 is an event log of a loan application process of a Dutch financial institute. It consists of 561761 events from 31509 cases.

The trivially identifiable elements of the log are case identifier, activity name and the timestamps. For the other columns, domain knowledge and the presence or absence of a value in a certain column can help to determine whether a column can be affiliated with a case, an event or even with some other entity of the process. However, in general case identifier and activity name can be freely chosen among the attributes. This enables process discovery techniques to generate different perspectives of the process while keeping the same event log structure. For our case study we choose the case identifier "case" and activity name "event" by interpreting the column naming and the respective cell values. Table 3.1 gives an overview how assigned we matched the event log characteristics to the columns of the case study log.

The identification of timestamps in this data set has been rather trivial. We have a "startTime" and a "completeTime" column indicating the start and end of an activity. For the decision whether a column should be declared an event attribute or a case attribute, we had to take a closer look at the data as we did for the case identifier and the activity name. To classify a column as a case attribute, a column must...
Using Graph Data Structures for Event Logs

<table>
<thead>
<tr>
<th>#</th>
<th>Column name</th>
<th>Data type</th>
<th>Event log element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>case</td>
<td>String</td>
<td>Case identifier</td>
</tr>
<tr>
<td>2</td>
<td>event</td>
<td>String</td>
<td>Activity name</td>
</tr>
<tr>
<td>3</td>
<td>startTime</td>
<td>Datetime</td>
<td>Event timestamp</td>
</tr>
<tr>
<td>4</td>
<td>completeTime</td>
<td>Datetime</td>
<td>Event timestamp</td>
</tr>
<tr>
<td>5</td>
<td>LoanGoal</td>
<td>Integer</td>
<td>Case attribute</td>
</tr>
<tr>
<td>6</td>
<td>ApplicationType</td>
<td>Integer</td>
<td>Case attribute</td>
</tr>
<tr>
<td>7</td>
<td>RequestedAmount</td>
<td>Float</td>
<td>Case attribute</td>
</tr>
<tr>
<td>8</td>
<td>Accepted</td>
<td>Boolean</td>
<td>Case attribute</td>
</tr>
<tr>
<td>9</td>
<td>Action</td>
<td>String</td>
<td>Event attribute</td>
</tr>
<tr>
<td>10</td>
<td>FirstWithdrawalAmount</td>
<td>Float</td>
<td>Event attribute</td>
</tr>
<tr>
<td>11</td>
<td>NumberOfTerms</td>
<td>Integer</td>
<td>Event attribute</td>
</tr>
<tr>
<td>12</td>
<td>OfferID</td>
<td>String</td>
<td>Event attribute</td>
</tr>
<tr>
<td>13</td>
<td>org:resource</td>
<td>String</td>
<td>Event attribute</td>
</tr>
<tr>
<td>14</td>
<td>MonthlyCost</td>
<td>Float</td>
<td>Event attribute</td>
</tr>
<tr>
<td>15</td>
<td>EventOrigin</td>
<td>String</td>
<td>Event attribute</td>
</tr>
<tr>
<td>16</td>
<td>EventID</td>
<td>String</td>
<td>Event attribute</td>
</tr>
<tr>
<td>17</td>
<td>Selected</td>
<td>Boolean</td>
<td>Event attribute</td>
</tr>
<tr>
<td>18</td>
<td>CreditScore</td>
<td>Integer</td>
<td>Event attribute</td>
</tr>
<tr>
<td>19</td>
<td>OfferedAmount</td>
<td>Float</td>
<td>Event attribute</td>
</tr>
</tbody>
</table>

Table 3.1: BPI Challenge 2017 Event Log Overview

contain the same values for all events (rows) of the same case. All other columns would then just be classified as event attributes. However, by taking a closer look at each column it becomes clear that the distinction of event and case attributes is not that trivial. The "Accepted" column for example, has Boolean datatype and will only receive a value after the activity (event, row 2 in table 3.1) A_Accepted which usually isn’t the first activity of an application (case, row 1) and thus has empty cells for the first activities of most of the cases. The "Accepted" column is likely to be a flag for whether an application got accepted by the financial institute or not and hence should rather be a case attribute. Furthermore, the column "EventOrigin" (row 15) indicates an event’s affiliation to one of the different entities of the process, i.e. Workflow, Application and Offer. With the help of this column we can distinguish the event attributes according to their respective entities. Rows 9, 10, 12, 14, 17, 18 and 19 of in table 3.1 contain information about offers. For this process multiple offers can be created for one application which we can distinguish by their "OfferID" (row 12). The "EventOrigin" value Application refers to the application and the Workflow events enrich the cases with workflow information such that per process instance exactly one application and exactly one workflow exists. Row 13 contains the resource that has been recorded for a certain event. This information is part of many event logs and enables us to apply organizational mining techniques as described in [1].
3.3 Insights

In this section we specify some desired insights that we typically want to derive from an event log. Some are rather generic insights that can be retrieved from any event log as described in section 2.1 and others can be considered more specific to the information contained in BPIC17 log, such as offers, as described in section 3.2.

1. Query a specific attribute of a specific event.
2. Query a specific attribute of a specific case.
3. Directly follows (DF) relations of events - Event X directly follows event Y in the same case without any events in between them.
4. Eventually follows relationships of events - Event X is eventually followed by event Y in the same case.
5. Case variants - The variant of a case is defined by the sequence of its activities. All cases with the same sequence of activities have the same variant.
6. Handover-of-work (HoW) social network - The resources of two events with directly follows relationship have a handover of work relationship. A social network can be created from the collection of all HoW relationships in the log.
7. Distance between two specific Activities - Distance can be time or activity count between two events with specific activities in one case.
8. Query a sequence of activities which depends on attributes of multiple entities of one case.

For every insight we will define at least one question specific to our case study to be answered in the evaluation section.
4 Representing Event Logs as Graphs

4.1 From Table to Graph

When creating a graph data structure from tabular data, there are multiple ways how the data can be represented. The design of the data structure depends on the purpose the graph is created for. To structure the design decisions for the graph we introduce a simple logic shown in figure 4.1. As discussed in earlier sections, the central elements of an event log are events and cases. Thus we create one node for every case identifier and one node for every event, i.e. row in the log. Additionally there might be other process-related entity identifiers we can derive from the log such as "org:resource" or "OfferID" in our case study as discussed in section 3.2. All other columns might become a property of either of the three node types or just be ignored if deemed irrelevant. However, by only following this generic logic for our node definition we could miss important information. Furthermore we want to take a closer look at the "EventOrigin" column that has also been discussed in section 3.2 already. Next to "Offer", this column indicates the existence of an "Application" and a "Workflow" entity and the latter two don’t have discrete identifiers other than the case identifier. At a first glance it might be counter intuitive to establish a distinct "Application" node type next to what we defined as a case in the loan application process. Consider the "Application" as some form that must be filed to start a case, whereas the case as we defined it is actually a combination of "Application", "Offer" and "Workflow" events. This way we can create relationships dedicated for the different aspects of the process more easily.
Table 4.1 shows an overview of the nodes and their properties as discussed above. The *-markings indicate the unique identifier of the respective nodes as property from the event data. Because the log contains exactly one (:Application) and exactly one (:Workflow) per case and also due to the lack of a dedicated identifier for these entities, we reused the case identifier. Note that upon creation of a new node or relationship, Neo4j creates an internal unique ID for it. All other properties have been derived from the logic above. Due to the (:Application) being the main subject of our definition of a (:Case) we decided to turn the case attributes to properties of both node types.

To complete the graph event log we need to create relationships between the new nodes. Table A.1 gives an overview of the relationships we used in the relationship syntax of Cypher which we introduced in section 2.3 with the labels of end and start node. Except for the relationship properties which, for better readability, are listed in a separate column.
Using Graph Data Structures for Event Logs

<table>
<thead>
<tr>
<th>#</th>
<th>Label</th>
<th>Properties as Column Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(:Case)</td>
<td>case* LoanGoal ApplicationType RequestedAmount</td>
</tr>
<tr>
<td>2</td>
<td>(:Event)</td>
<td>EventID* EventOrigin startTime completeTime event EventOrigin Action</td>
</tr>
<tr>
<td>3</td>
<td>(:Resource)</td>
<td>org:resource*</td>
</tr>
<tr>
<td>4</td>
<td>(:Offer)</td>
<td>OfferID* MonthlyCost NumberOfTerms OfferedAmount CreditScore</td>
</tr>
<tr>
<td>5</td>
<td>(:Application)</td>
<td>case* LoanGoal ApplicationType RequestedAmount</td>
</tr>
<tr>
<td>6</td>
<td>(:Workflow)</td>
<td>case*</td>
</tr>
</tbody>
</table>

Table 4.1: Graph event log nodes

<table>
<thead>
<tr>
<th>#</th>
<th>Relationship</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(:Event) -[:EVENT_TO_CASE] -&gt; (:Case)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(:Event) -[:DF] -&gt; (:Event)</td>
<td>Duration between the events</td>
</tr>
<tr>
<td>3</td>
<td>(:Resource) -[:RESOURCE_TO_EVENT] -&gt; (:Event)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(:Offer) -[:OFFER_TO_CASE] -&gt; (:Case)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(:Event) -[:EVENT_TO_OFFER] -&gt; (:Offer)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(:Event) -[:EVENT_TO_APPLICATION] -&gt; (:Application)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(:Application) -[:APPLICATION_TO_CASE] -&gt; (:Case)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(:Event) -[:EVENT_TO_WORKFLOW] -&gt; (:Workflow)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(:Workflow) -[:WORKFLOW_TO_CASE] -&gt; (:Case)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(:Event) -[:O_DF] -&gt; (:Event)</td>
<td>Duration between the events</td>
</tr>
<tr>
<td>11</td>
<td>(:Event) -[:A_DF] -&gt; (:Event)</td>
<td>Duration between the events</td>
</tr>
<tr>
<td>12</td>
<td>(:Event) -[:W_DF] -&gt; (:Event)</td>
<td>Duration between the events</td>
</tr>
<tr>
<td>13</td>
<td>(:Resource) -[:HOW] -&gt; (:Resource)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>(:Resource) -[:O_HOW] -&gt; (:Resource)</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(:Resource) -[:A_HOW] -&gt; (:Resource)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(:Resource) -[:W_HOW] -&gt; (:Resource)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Graph event log relationships

The identification of relationships has been based on the event log characteristics and the insights defined in the previous chapter. The relationship names have been chosen in a way that they should be self-explanatory even without the context of their start and end node type. Table A.1 shows the list of
relationships that we retrieved from the event log. Relationships 1 and 2 are generic enough to be applicable to any graph event log derived from a sequential event log. This is due to the fact that they represent mandatory event log characteristics as relationship 1 associates events with a case and relationship 2 realizes the temporal ordering of events within a case by introducing the directly follows (DF) relationships between the events of a case. Whereas the other relationships depend on the information contained in the individual logs. For example relationship 4 would of course require resource information to be contained in the log.

Where sequential event logs are limited to their case internal order of events and rely on selection and projection to show that data columns belongs to specific instances such as an offer, in a graph-based event log these relationships can be added to the log itself. The -[:O_DF]-> for example introduces the directly follows relationship for events of offers. Relationship 9 represents a similar relationship, but for events that belong to the "Application" entity only and not to the case, as we discussed in the paragraph for the node creation earlier. With the help of these distinct types of DF relationships we enable views on different types of entities within the log. For example if an event log allows a creation of a "global" hand-over of work (HOW) view, i.e. contains resource information, that shows which resources perform activities before/after one another, we can now narrow this perspective down to the scope of one offer instance just by using the -[:O_HOW]-> relationship. To improve the readability of our graph and queries, we simplified "Application", "Offer" and "Workflow" to "A", "O" and "W" respectively.

4.2 Data Preparation

The BPI Challenge 2017 comes in .xes format which is not directly supported by Neo4j import functionality, whereas .csv is. Therefore ProM was used to convert the .xes to .csv. The resulting file was then directly loaded by Neo4j or Python for preparation.

Through the course of our research, we realized that the conversion from .xes to .csv format by ProM Light (1.2) had led to some unexpected irregularity in the data as can be seen in appendix A.1. We verified that the combination of "O_Create Offer" directly followed by "O_Created" could reliably determine the attributes to an offer for all inspected cases in a sample set. Under the assumption that this holds for the entire dataset we have been able to make use of this irregularity to assign the offer attribute values to the correct offer identifiers. Furthermore, the converted .csv file appeared to have a quite high number of duplicates. This we cured by "merging" identical rows.

There are two main ways to import data in Neo4j: by using the database API or by import from a .csv file. While the assessment of which of the two ways is be better is out of the scope of this research, they have been tested by creating a simple (:Case) <-[:EVENT_TO_CASE]- (:Event timestamp) structured graph event log. The API has been tested using the py2neo package for python and indicated slower import speed. Thus we opted for the Neo4j csv import functionality which allows to import the csv file and query it with Cypher right in the built-in Neo4j Browser.

To enable a fast result validation, we selected a subset of 20 cases of the BPIC17. We chose a subset containing cases of the same variant and cases with one and two offers to account for the concepts of process entities. Please check appendix A for the list of case IDs contained in our sample set.

Furthermore, the determination of -[:O_DF]->, -[:A_DF]-> and -[:W_DF]-> required some internal ordering index of the respective instance. For example, if one case has two offers, the log will get two different indices for the offers and one for the case itself. The case index is incremented for all events of a case in the order they appear in the log, whereas the index for the offers will only increase for events with "EventOrigin" = "Offer" and for every offer ID in a case there is a different index. For the workflow
and application entities this applies accordingly. A sample case is illustrated in appendix A.3.

As a final preparation step for the Neo4j import, we split the timestamp data into their atomic values year, month, day, hour, minute, second and microsecond. This way we could import the timestamps as "LocalDateTime" datatype in Neo4j which enables us to use temporal functions, e.g. to calculate the duration of activities or the time between two events. Even though a reformattting would also be sufficient for such an import, we decided to go for the "atomic" method since it gives us more control over the different values added to the "LocalDateTime" objects and it appears to be less error prone.

The resulting sample log includes 20 cases, 349 events 22 activities, 50 resources and 25 offers.

### 4.3 Graph Creation

To create the graph according the the design in section 4.1 we used a sequence of Cypher queries to import the prepared data to Neo4j. The queries can be found in appendix B. Since some of the queries rely on the output of other queries, we recommend to keep the order of the listing in the appendix for replication. These queries have been used for both, creating the graph from the sample and from the full log.

Figure 4.2 shows the schema of the graph event log.

![Graph DB Schema](image)

Figure 4.2: Graph DB Schema

The sample graph has been created using a machine with an Intel i7 CPU @ 2.8 GHz with 16 GB of memory. The query that took the longest time to complete was query 3 of appendix B with 0.339 seconds and the cumulative execution time of all queries is 1.215 seconds. No adjustments of the queries or the DB settings were needed to complete sample graph creation. The implementation with our sample set
contains 484 nodes with 6 different node types and 2065 relationships with 16 different relationship types.

Creating the graph from the full log was also successful, but needed some adjustments on the database settings and the queries. In a first step we created unique constraints for the name properties (IDs) of :Event, :Case and :Resource nodes. This does not only ensure uniqueness of these nodes, but also creates indices leading to a better performance of our queries. Using the same machine we used for the sample, the longest query execution took 38.875 seconds and the total execution time of all queries summed up to 156.679 seconds. In this case query 2 had the longest execution time. The initial queries we defined for the graph creation (not included in the report) was less efficient and took a cumulative execution time of roughly 75 minutes. We have been able to significantly improve the query performance by reducing the load csv operations, particularly for query 6 to create the -[:EVENT_TO_OFFER]->, by adding the "OfferID" as property to event nodes. The downside of this workaround is that we temporarily write redundant information to the graph, i.e. the "OfferID" and the offer index as properties of (:Event) and (:Offer) nodes such that we can read the required data from the graph instead of the csv file. The temporary data is cleaned up by query 20 taking 1.394 seconds. It was necessary to add "USING PERIODIC COMMIT 1000" as first line to queries 1,2,3 and 5 as shown in appendix B.2 to reduce the memory size needed for the execution. Furthermore the default value of 1 GB for memory usage of the Neo4j DB ("dbms.memory.heap.max_size" parameter) prevented some queries to complete. Increasing the option to 12 GB enabled all queries to finish successfully without running into memory problems for the full log. Less than 12 GB might also be sufficient, but we did not conduct further testing.

The full graph event log has 699338 nodes with 6 different node types and 2811907 relationships with 16 different relationship types.

Note that we solely used Cypher queries to create the relationships in table A.1. No additional algorithms have been needed, which is usually the case for example to create handover-of-work social networks based on sequential event logs.
5 Quering Graph-based event logs using Cypher

In section 3.3 we identified a set of desired insights we want to get from the log. From each of these insights, a specific question on the data will be derived and get answered by querying the graph structure introduced in chapter 4. In order to evaluate the results, the same question will be answered by analysing the sequential event log with Python or common process mining tools such as ProM or Disco. Note that the validation of correctness of the queries was done with the same sample cases (see appendix A) of the BPIC17 log except for question 12, where the full log was used to check the results for correctness. All queries have also been executed on the full graph to ensure they actually can be executed and to give an indication on how they perform on the full data. So except for question 12 every query is checked for correctness on the sample graph and for performance/feasibility on the full graph.

The case study on the BPIC17 log has been conducted using Neo4j version 3.5.4 using an Intel i7 CPU @ 2.8 GHz with 16 GB of memory.

Depending on the query, Neo4j offers different representations of results, e.g. graph structure similar to figure 2.1 or table format. Therefore, the screenshots might differ per query result. Part (a) of the screenshots always represents the Neo4j output and part (b) the output of the respective tool used on the sequential log.

5.1 Query Attributes of an Events

**Question 1:** What is the completion time of the "A_Submitted" activity of the case "Application_681547497"?

```
MATCH (c:Case) ←[:EVENT_TO_CASE]→ (e:Event)
WHERE c.name = "Application_681547497" AND e.activity = "A_Submitted"
RETURN e.completetime
```

The output of the query is shown in figure 5.1a (a). Since we only returned a property, the Neo4j output cannot be visualized as a graph. Figure 5.1b shows the corresponding activity in Disco.

![Figure 5.1: Question 1 Screenshots](image)

Note the query returns as many lines as "A_Submitted" activities exist in the case.
We also want to demonstrate that the attributes originally defined as event attributes in Table 3.1 have partly been translated to nodes as defined in Table 4.1 and some of the other event attributes became attributes of the entity itself, e.g. an offer as entity and the offered amount as node property in the graph: (:Offer {OfferedAmount}). The following query shows that the offered amount can still be queried, but in a different way compared to a property of an :Event node as shown in the query above.

The execution time on the full graph was 0.002 seconds.

Question 2: What are the offered amounts in case "Application_681547497"?

```
MATCH (c:Case) <-[:OFFER_TO_CASE]- (o:Offer)
WHERE c.name = "Application_681547497"
RETURN o.name, o.offeredamount
```

Since we have two offers in this case, we get two rows back from the query in Neo4j as can be seen at figure 5.2a. Offers "Offer_716078829" and "Offer_897102764" have both been created for case "Application_681547497" and both with an offered amount of 25000. As we can see on the Disco screenshot in figure 5.2b, we need to make the assumption that "O_Create Offer" directly followed by "O_Created" in the sequential data belong to the same offer to relate an OfferID to its attributes.

The execution time on the full graph was 0.003 seconds.

5.2 Query Attributes of Cases

Querying case attributes works similar to querying event attributes.

Question 3: What was the requested amount in case "Application_681547497"?

```
MATCH (c:Case)
WHERE c.name = "Application_681547497"
RETURN c.requestedamount
```
Besides the fact that we queried a different type of node and property name, there is no difference to querying an event and thus the execution time on the full graph was with 0.003 seconds equally low.

## 5.3 Directly follows relations of events

Here we are going to query the graph for one of the fundamental characteristics of a sequential event log: the directly follows relationships between events. In order to do so, we select the "A_Submitted" activity of case "Application_681547497" we have seen in question 1 already and check for its directly following activity.

**Question 4**: What activity follows "A_Submitted" in case "Application_681547497"?

```sql
MATCH (c:Case) <-[:EVENT_TO_CASE]-(e1:Event) <-[:DF]-(e2:Event)
WHERE c.name = "Application_681547497" AND e1.activity = "A_Submitted"
RETURN e1, e2
```

For question 4 we can choose the graph visualization, because the return statement consists of graph elements and not only of properties. Figure 5.4 shows that "A_Concept" directly follows "A_Submitted". The standard visualization of Neo4j Browser shows all relationships between the nodes returned by the query. This is why the A_DF relationship is also shown in the screenshot.

The execution time on the full graph was 0.009 seconds.

## 5.4 Eventually follows relationships of events

We now want to see if there is a directly follows relationship between two events with specific activities in case "Application_681547497".

**Question 5**: Does "A_Accepted" eventually follow "A_Submitted" in case "Application_681547497"?

For this question we don’t know how many relationships and nodes need to be walked along the path between "A_Accepted" and "A_Submitted". The only thing we certainly know is that if there is an eventually follows relationship between the two activities, the path will consist of an unknown number of event nodes and DF relationships. For this type of queries we can use the *-operator for the DF
relationship. The *-operator considers an arbitrary number of DF relationships to be matched between the event nodes in the MATCH clause.

1. MATCH (c:Case) <-[:EVENT_TO_CASE]-(e1:Event) <-[:DF*]- (e2:Event)
2. WHERE c.name = "Application_681547497" AND e1.activity = "A_Submitted" AND e2.
   activity = "A_Accepted"
3. RETURN e1, e2

Figure 5.5: Question 5 Neo4j Screenshot

When only checking for the existence of the eventually follows relationship in question, we get nodes that are not necessarily directly connected, as can be seen in figure 5.5. In this case we only know that they are connected, but the query result does not show how. If we didn’t have a matching pattern the result would be empty. The query took 0.022 seconds on the full graph.

To see the respective path(s) between the nodes we are interested in, we can alternatively issue:

1. MATCH (c:Case) <-[:EVENT_TO_CASE]-(e1:Event) <-[:DF*]- (e2:Event)
2. WHERE c.name = "Application_681547497" AND e1.activity = "A_Submitted" AND e2.
   activity = "A_Accepted"
3. RETURN (e1:Event) <-[:DF*]- (e2:Event)

Which took 0.011 seconds on the full graph.

Figure 5.6: Question 5 Neo4j Screenshot with Path

5.5 Case variants

Now let’s consider the case variants.

We might want to as what variant case "Application_681547497" has. The following query completed in 0.006 seconds on the full graph.
MATCH (c:Case)
WHERE c.name = 'Application_68154797'
RETURN c.variant

The answer is variant number 8 for the sample graph. However, since the variant indices will differ from tool to tool and thus are hard to compare. Therefore let us check for the variant with the most traces in the sample.

Question 6: How many traces has the most frequent variant?

MATCH (c:Case)
WITH c.variant as variant, count(*) AS count
ORDER BY count DESC
LIMIT 1
MATCH (c:Case)
WHERE c.variant = variant
RETURN c.name, variant, count

Figure 5.7: Question 6 Neo4j Screenshot

This shows that the most frequent variant occurs three times in the sample. The screenshot in figure 5.7 and the Disco screenshot in figure C.6 in appendix C show the same case three case ids for the most frequent trace variant. The query completed after 0.237 seconds on the full graph.

5.6 Handover of Work

Now we want to get a handover-of-work social network of our resources.

Question 7: How does the handover-of-work social network look like for the sample set?

MATCH p = (r1)-[:HOW]->(r2)
RETURN p

This returns all handovers of work between resource including selfloops. The resulting graph was compared with the result of the "Mine for a Handover-of-Work Social Network" plugin of ProM 6.8. Since the results are too large to be captured on a single screenshots, we compared parts of them to check if the results correspond. The screenshots can be found in figure C.7 in appendix C. The two screenshots show similar results. The ProM graphs only shows the HOW relationships without self loops, i.e. users performing several consecutive activities. The Neo4j graph contains self loops, but by adding a "WHERE r1.name <> r2.name" clause the query can be adjusted accordingly. We considered the self loops useful information, especially if we take frequencies into account as can be seen in the next query. The graph visualisation of Neo4j also shows all other relationships between the nodes in the results. For better readability the actual HOW relationships are drawn slightly bigger than the others. Please note that this is just a feature of the graph view and the result data does not contain other relationships than HOW. The query took 1.227 seconds to complete on the full graph.

We can add frequencies only by taking the DF relations into account, because HOW relations only indicate that there is a handover, but not how often. We can formulate the following query to take the frequency into account:
MATCH (r1:Resource)-[:RESOURCE_TO_EVENT]->(e1:Event)<-[DF]-(e2:Event)<-[RESOURCE_TOEvento2:Resource)
RETURN r1 AS from, r2 AS to, count(*) AS frequency
ORDER BY frequency DESC

Note that the count(*) adds the amount of handovers to the table representation of the Neo4j output. The most frequent items of the result for the sample set can be inspected in appendix C.8. Interestingly the most frequent HOW relationships are self loops. The execution time on the full graph was 1.871 seconds.

5.7 Handover of Work based on Entities

Now we want to assess a more log-specific view on the handover-of-work domain. Now we only want to consider handovers within offers. Since offers are no typical element of an event log, it is a specific insight to BPIC17.

Question 8: How does the order-based handover-of-work social network look like for the sample set?

MATCH p =(r1)-[:O_HOW]->(r2)
RETURN p

Like in section 5.6, we again compared a sub-graph of the results in Neo4j and ProM. The screenshots of the ProM and Neo4j graphs can be found in figure C.9. The network mined by ProM required loading the sample log with "OfferID" as case identifier and subsequent filtering of non offer related events prior to the application of the "Mine for a Handover-of-Work Social Network" plugin. Again, except for the self loops, the O_HOW relationships of both graphs are equal. The query took 0.83 seconds to complete on the full graph. Again we can extend the query to take handover frequencies into account:

MATCH (r1:Resource)-[:RESOURCE_TO_EVENT]->(e1:Event)<-[O_DF]-(e2:Event)<-[RESOURCE_TOEvento2:Resource)
RETURN r1 AS from, r2 AS to, count(*) AS frequency
ORDER BY frequency DESC

A partial output for the sample graph can be found in appendix C.10. Similar to the HOW social network, the self loops represent the most frequent O_HOW relationships. The execution time on the full graph was 1.162 seconds.

5.8 Distance between two specific Activities

Here we want to see how long the shortest distance between the creation of an offer and its acceptance is. For the first question to this insight, we consider the distance to be the time between the completion of "O_Created" and the start of "O_Accepted" of an offer.

Question 9: What is the duration of the longest accepted offer in terms of time?

MATCH (start:Event{activity:"O_Created"})-[:EVENT_TO_OFFER]->(o:Offer)<-[EVENT_TO_OFFERo:Offer]<-[O_Offer]- (end:Event{activity:"O_Accepted"})
WITH start, end, duration.between(start.completetime, end.starttime) AS time, o
ORDER BY duration.between(start.completetime, end.starttime) DESC
LIMIT 1
RETURN start, end, time.days AS days, (toFloat(time.minutes)/60) AS hours, o

The query could be less complex, but the "duration.between()" function returns a rather unusual format, "P0M24DT53003.990000000S" in this case. For better readability the output has been reshaped. Figure C.11 shows the results of the query and a corresponding view in Disco. When rounding up the Neo4j result we get the exact same duration shown in the disco view for the same offer.
The query took 0.888 seconds to complete on the full graph.

Question 10: What is the duration of the longest accepted offer in terms of activities?

The length function with a path variable as input returns the "hops" between nodes in the path. The query returns 20, thus the longest sequence of activities from "O_Created" to "O_Accepted" in the sample set consists of 21 activities including the two. By counting the sequence of activities in the Disco screenshot in figure C.12 we also get 21 activities including the two. The query took 2.744 seconds to complete on the full graph.

We can also reduce this question to consider the activities belonging to the offers, similar to question 8 but in the context of duration.

Question 11: What is the duration of the longest accepted offer in terms of offer-specific activities?

The result for this query is 3, thus 4 activities. Taking a closer look revealed that every accepted offer in our sample set took exactly 4 order-related activities from "O_Created" to "O_Accepted". The Neo4j result can again be verified by counting the order related activities in figure C.13(b).

The execution time on the full graph was 0.93 seconds. The query on the full graph revealed that every accepted offer in the data set consists of maximum 4 activities.

5.9 Involve multiple Entities

With the last question we want analyse the data for very log specific information. We want to retrieve sequences of activities (paths) per entity of a case, but only for those cases that meet our requirements.

Question 12: What are the paths between "A_Create Application" and "O_Cancelled" for those cases with at least two different offers with "O_Created" directly followed by "O_Cancelled" on the entity level.

This is a typical question that can be considered challenging for today's standard process mining tools using sequential event logs. It requires several steps where the log needs to be analysed and filtered for the different components of the question. Here we need to identify all offers "O_Created" directly followed by "O_Cancelled" on the entity (offer) level. We also need to identify all cases we are interested in, i.e. having at least two of these offers. Only then we can start answering the question and retrieve the sequences of activities between "A_Create Application" and "O_Cancelled" for the individual offers.

Due to the low number of cases that have the desired characteristics, we evaluated this question also for correctness on the full graph.

The following query returns the paths we are looking for:
MATCH (:Event {activity: "O_Created"}) <-[:O_DF]-(e:Event {activity: "O_Cancelled"}) <-[:EVENT_TO_OFFER]-(o:Offer) <-[:OFFER_TO_CASE]-(c)
WITH e AS O_Cancelled
MATCH p = (A_Created:Event {activity: "A_Create Application"}) <-[:DF*]-(O_Cancelled:Event {activity: "O_Cancelled"})
RETURN p

This query took 0.406 seconds to complete on the full graph when using Neo4j Browser. It is a combination of three different queries where the subsequent MATCH clauses can use the output of the preceding selection clauses. The MATCH clause in line 4 can match all cases we are interested in by using the variable "c" and hands the "O_Cancelled" events of the desired offers over to the final query. The final query then returns the paths (DF) from "A_Create Application" to "O_Cancelled" by using the *-operator which gives us the desired result. We get 218 sequences returned, which means we have 218 offers in our output. If we want to include the cases in our output, we can add the "c" variable to the WITH clause in line 5 and include the case in the last part of our query as follows:

MATCH (e1:Event {activity: "O_Created"}) <-[:O_DF]-(e:Event {activity: "O_Cancelled"}) <-[:EVENT_TO_OFFER]-(o:Offer) <-[:OFFER_TO_CASE]-(c:Case)
WITH c AS c, count(o) AS ct
WHERE ct > 1
MATCH (:Event {activity: "O_Created"}) <-[:O_DF]-(e:Event {activity: "O_Cancelled"}) <-[:EVENT_TO_OFFER]-(o:Offer) <-[:OFFER_TO_CASE]-(c)
WITH c AS c, e AS O_Cancelled, o AS o
MATCH p = (A_Created:Event {activity: "A_Create Application"}) <-[:DF*]-(O_Cancelled:Event {activity: "O_Cancelled"}), (O_Cancelled) <-[:EVENT_TO_CASE]-(c)
RETURN p,c,o

This query, when run in Neo4j Browser, took 0.455 seconds to complete on the full graph.

Due to the complexity of this query the evaluation of the sequential log has been conducted in Python. The script can be found in appendix C.2.1. The results of both, the CSV analysis and the graph analysis have been included in the script to enable a fast and sound comparison of both outputs. The graph analysis used the latter query to include the cases and both results have been transformed to the same data shape (case id, offer id and the sequence of activities) for better comparability. As we can see in the output of the script (appendix C.2.2) the full data set contains 103 cases with 218 offers with the characteristics in question and both outputs are equal.

Remarkably, the CSV analysis took more than 15 minutes whereas the graph analyses took less than a second on an Intel i7 CPU @ 2.8 GHz with 16 GB of memory with Python. However, a performance comparison is out of scope of this research and both the CSV and graph analysis have not been optimized for efficiency.
6 Conclusion

In this report we have been able to explore a way to represent and query event log data in a graph and we have been able to show that a graph DB and more specific Neo4j is an appropriate way to represent an event log for process mining. To conclude the report we want to verify if we have been able to answer the given research questions:

1. **How can the concepts of event logs be represented in graph databases?**

   We have shown one possible way to represent an event log as property graph. The concepts case, event and activity are represented as property graph elements as well as resources and timestamps. We introduced entities as distinct nodes to make querying the graph on the different process aspects easier. Also the introduction of entity specific directly follows relations, such as O\_DF, enabled us to define simple queries to analyse questions that require different filters and projections or special algorithms with sequential event data. In terms of the graph design, many times the question was not "can we represent the data in a graph for our purpose?" but rather "what is the best way to represent the data in a graph for our purpose?". Because property graphs offer different ways to represent data and our main goal is process mining we combined the standard concepts of sequential event logs and augmented them with additional perspectives such as entities by only using the data available in the sequential log.

   a) **How can cases and the temporal and causal ordering of events be represented in graphs?**

      By introducing different types of relationships between the nodes we have been able to model causal and temporal relationships of events, e.g. with E\_DF relationship. We even could enhance the view by making orders a distinct "entity" that is related to a case, compared to the rather fuzzy representation in a sequential event log where it is not clear whether this is an event attribute or a case attribute.

   b) **How can attributes and values be stored and shared across cases?**

      We successfully showed that attributes and their values can be stored as both, nodes or properties to nodes and both ways enable the graph to be queried these attributes over multiple cases.

2. **How to use graph-based queries to answer process analysis questions within and across cases?**

   Chapter 5 was dedicated to demonstrate answering process analysis questions. Starting from simple tasks like querying an attribute value of a specific event or case, we went on to more elaborate questions across cases. We have also been able to query a handover-of-work social network with a single query.

   The only shortcoming we could identify, if it can be considered a shortcoming, is the longer total processing times when importing a full log into Neo4j and creating and configuring the database compared to loading the full sequential log into ProM for example. On the other hand, the import usually only takes place once and does not require reloading in order to create new data perspectives. In terms of data representation or retrieval, we have not been facing any issues. We could retrieve the same results compared to the traditional process mining tooling for all the queries in this report. Most remarkably we have been able to perform the complete graph creation and graph analysis on DB level, i.e. by only using Cypher queries. Even though it does not seem likely that all possible process mining questions on graph event data can be answered by solely using Cypher, having a graph database as source for process mining applications can enable deeper insights on processes. For pure data analysis related queries it seems likely to be possible to answer all process mining related queries on BPIC17 based graph design.
of this report. For some questions it will even be harder to evaluate the queries by means of conventional process mining tools as we experienced for question 12 in chapter 5. For more sophisticated applications like predictive analytics, for example to predict the remaining cycle time of a case, where mathematical models need to be trained Cypher and Neo4j will not of course suffice, but still can serve as data source.

This report is the first step into a new area in process mining, thus there are manifold research directions that may follow on. For example, a systematic approach for translating sequential event logs to graphs or the creation of a graph event logs for process mining might be required to form the base of new process mining techniques. Graph event logs could for example be extended with more context to the resource perspective, e.g. schedules, to take them into account when assessing process performance. Graph event logs might also be able to contain events of multiple processes that possibly interact, say different processes of the same information system, to again widen the perspective on the processes compared to the isolated process that is usually captured by a sequential event log.
A Data Preparation

A.1 Data Conversion Problem

The following two screenshots show how some of the data got confused by the conversion.

Figure A.1: Offer Information in the .xes File

Figure A.2: Offer Information in the .csv File
## A.2 Sample Set

<table>
<thead>
<tr>
<th>#</th>
<th>Case ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Application_2045572635</td>
</tr>
<tr>
<td>2</td>
<td>Application_2014483796</td>
</tr>
<tr>
<td>3</td>
<td>Application_1973871032</td>
</tr>
<tr>
<td>4</td>
<td>Application_1389621581</td>
</tr>
<tr>
<td>5</td>
<td>Application_1564472847</td>
</tr>
<tr>
<td>6</td>
<td>Application_430577010</td>
</tr>
<tr>
<td>7</td>
<td>Application_889180637</td>
</tr>
<tr>
<td>8</td>
<td>Application_1065734594</td>
</tr>
<tr>
<td>9</td>
<td>Application_681547497</td>
</tr>
<tr>
<td>10</td>
<td>Application_1020381296</td>
</tr>
<tr>
<td>11</td>
<td>Application_180427873</td>
</tr>
<tr>
<td>12</td>
<td>Application_2103964126</td>
</tr>
<tr>
<td>13</td>
<td>Application_55972649</td>
</tr>
<tr>
<td>14</td>
<td>Application_1076724533</td>
</tr>
<tr>
<td>15</td>
<td>Application_1639247005</td>
</tr>
<tr>
<td>16</td>
<td>Application_1465025013</td>
</tr>
<tr>
<td>17</td>
<td>Application_1244956957</td>
</tr>
<tr>
<td>18</td>
<td>Application_1974117177</td>
</tr>
<tr>
<td>19</td>
<td>Application_797323371</td>
</tr>
<tr>
<td>20</td>
<td>Application_1631297810</td>
</tr>
</tbody>
</table>

Table A.1: Graph event log relationships
A.3 Instance Indices

![Figure A.3: Indices for the different Instances](image)

A.4 Python Script for Data Preparation

The Python script used for the data preparation of the sample. The same script has been used for the preparation of the full dataset by using "sampleIds = cases".

```python
import pandas as pd

# load full log
loan_raw = pd.read_csv('bpiChallenge17.csv', keep_default_na=False)

# remove duplicates from the dataset
loan_raw.drop_duplicates(keep='first', inplace=True)

# renew the index to close gaps of removed duplicates
loan_raw = loan_raw.reset_index(drop=True)

# create a list of all cases in the dataset
cases = loan_raw['case'].unique().tolist()
noOfCases = len(cases)

# hard coded case IDs to ease replication
sampleIds = ['Application_2045572635',
             'Application_2014483796',
             'Application_1973871032',
             'Application_1389621581',
             'Application_1564472847',
             'Application_430577010',
             'Application_889180637',
             'Application_1065734594',
             'Application_681547497',
             'Application_1020381296',
             'Application_180427873']
```
sampleList = []  # create a list of lists for the sample data containing a list of events for each of the selected cases
variants = []  # helper list for the variants
for case in sampleIds:
    offers = []  # helper list for counting the events with eventOrigin = Offer
    applications = 0  # helper variable for counting the events with eventOrigin = Application
    workflows = 0  # helper variable for counting the events with eventOrigin = workflows
    variant = []  # helper list for the variant of the current case
    for index, row in loan_raw[loan_raw.case == case].iterrows():  # first iteration through the cases for variant detection
        if row["EventOrigin"] == "Workflow":  # add the variants list
            variants.append(row["event"])
        else:
            row["EventOrigin"] == "Offer" and "Offer_" in row["OfferID"];  # check if event has an offer ID and is an offer event
            offerID = loan_raw.loc[index+1]["OfferID"];  # assign the offerID of the next event (O_Created) to this activity
    if not variant in variants:  # if variant of the current case has not been observed before, add to the variants list
        variants.append(variant)
    for index, row in loan_raw[loan_raw.case == case].iterrows():  # second iteration through the cases for adding data
        if row["event"] == "O_Create Offer":  # this activity belongs to an offer but has no offer ID
            if loan_raw.loc[index+1]["event"] == "O_Created":  # if next activity is "O_Created" (always directly follows "O_Create Offer" [verified with Disco])
                offerID = loan_raw.loc[index+1]["OfferID"];  # assign the offerID of the next event (O_Created) to this activity
                row["OfferID"] = loan_raw.loc[index+1]["OfferID"];  # add the event data to rowList
                rowList = list(row)  # add the event data to rowList
                rowList.append(index)  # add global index for the sequence of events
                if row["EventOrigin"] == "Offer" and "Offer_" in row["OfferID"];  # check if event has an offer ID and is an offer event
                    offers.append(row["OfferID"]);  # add offerID entry to helper list
                    rowList.extend((offers.count(row["OfferID"]), 0, 0, "Variant_" + str(variants.index(variant)+1)));  # add order index for offer and variant
                    rowList.extend((0, applications, 0, "Variant_" + str(variants.index(variant)+1)));  # add order index for application and variant
                    rowList.extend((0, workflows, "Variant_" + str(variants.index(variant)+1)));  # add order index for workflow and variant
                    else:  # add -1 as default indices for easy detection of undesired outcomes
                        rowList.extend([-1, -1, -1, "Variant_" + str(variants.index(variant)+1))]).append(rowList)  # add the extended, single row to the sample dataset

sampleList.append(rowList)  # add the sample data

header = list(loan_raw)  # save the original header data
header = list(loan_raw)  # extend the header for the indices created for case, offer, application, workflow and variant
loan_samples = pd.DataFrame(sampleList, columns=header);  # create pandas dataframe and add the samples
loan_samples["startTime"] = pd.to_datetime(loan_samples["startTime"])
Using Graph Data Structures for Event Logs

```python
loan_samples['completeTime'] = pd.to_datetime(loan_samples['completeTime'])

# create separate columns (year/month/day/hour/minute/second/microsecond) for importing datetime values into neo4j as datetime object
loan_samples = loan_samples.assign(sY=loan_samples['startTime'].map(lambda x: x.year))
loan_samples = loan_samples.assign(sM=loan_samples['startTime'].map(lambda x: x.month))
loan_samples = loan_samples.assign(sD=loan_samples['startTime'].map(lambda x: x.day))
loan_samples = loan_samples.assign(sHH=loan_samples['startTime'].map(lambda x: x.hour))
loan_samples = loan_samples.assign(sMM=loan_samples['startTime'].map(lambda x: x.minute))
loan_samples = loan_samples.assign(sSS=loan_samples['startTime'].map(lambda x: x.second))
loan_samples = loan_samples.assign(sMS=loan_samples['startTime'].map(lambda x: x.microsecond))
loan_samples = loan_samples.assign(cY=loan_samples['completeTime'].map(lambda x: x.year))
loan_samples = loan_samples.assign(cM=loan_samples['completeTime'].map(lambda x: x.month))
loan_samples = loan_samples.assign(cHH=loan_samples['completeTime'].map(lambda x: x.hour))
loan_samples = loan_samples.assign(cMM=loan_samples['completeTime'].map(lambda x: x.minute))
loan_samples = loan_samples.assign(cSS=loan_samples['completeTime'].map(lambda x: x.second))
loan_samples = loan_samples.assign(cMS=loan_samples['completeTime'].map(lambda x: x.microsecond))

if not (any(loan_samples.offer_index == -1) or any(loan_samples.application_index == -1) or any(loan_samples.workflow_index == -1)):  # check if any event has an invalid index
    # write to file
    sampleFile = 'loan_sample.csv'
    loan_samples.to_csv(sampleFile, index=False)
    print('Data preparation successful – data written to file for Neo4j import')
else:
    print('NO FILE WRITTEN – DATAPREP NOT SUCCESSFUL')
```
B Implementation

The "loan_sample.csv" file must be placed in the import folder of the Neo4j DB you are working with. Comment lines denoted by a leading "//" contain the query number, a short description and the Neo4j output in parenthesis. The graph database has been constructed with the following queries:

B.1 Sample Graph

```sql
//1 create case nodes with case attributes (Added 20 labels, created 20 nodes, set 100 properties, completed after 274 ms.)
LOAD CSV WITH HEADERS FROM "file:///loan_sample.csv" as line
WITH line.case as case,
    line.LoanGoal as loanGoal,
    line.ApplicationType as applicationType,
    line-RequestedAmount as requestedAmount,
    line.FirstWithdrawalAmount as firstAmount,
    line.NumberOfTerms as n_Terms,
    line.variant_index as vIndex
MERGE (c:Case {name: case, variant: vIndex, loanGoal: loanGoal, applicationType: applicationType, requestedAmount: toInteger(requestedAmount)})

//2 create event nodes with event attributes and relationships to cases (Added 349 labels, created 349 nodes, set 3743 properties, created 349 relationships, completed after 274 ms.)
LOAD CSV WITH HEADERS FROM "file:///loan_sample.csv" as line
WITH line.case as case,
    line.EventID as event,
    line.EventOrigin as eventClass,
    line.event as activity,
    line.Action as action,
    line.startTime as startTime,
    line.completeTime as completeTime,
    line.offerID as offer,
    line.case_index as cIndex,
    line.offer_index as oIndex,
    line.application_index as aIndex,
    line.workflow_index as wIndex,
    line.sY as sY, line.sD as sD, line.sM as sM, line.sHH as sHH, line.sMM as sMM,
    line.sSS as sSS, line.sMS as sMS,
    line.cY as cY, line.cD as cD, line.cM as cM, line.cHH as cHH, line.cMM as cMM,
    line.cSS as cSS, line.cMS as cMS
MATCH (c:Case {name: case})
CREATE (e:Event {name: event, starttime: localdatetime({year:toInt(sY), month:toInt(sM), day:toInt(sD), hour:toInt(sHH), minute:toInt(sMM), second:toInt(sSS), microsecond:toInt(sMS)})})
CREATE (e) -[:EVENT_TO_CASE]-> (c)

//3 create nodes for each resource and edges to event nodes (Added 50 labels, created 50 nodes, set 50 properties, created 349 relationships, completed after 339 ms.)
LOAD CSV WITH HEADERS FROM "file:///loan_sample.csv" as line
WITH line.resource as resource, line.EventID as event
MATCH (e:Event {name: event})
```
Using Graph Data Structures for Event Logs

```sql
MERGE (r:Resource {name: resource})
CREATE (r) -[:RESOURCE_TO_EVENT]-> (e)

//4 create directly follows relationships for events (Table Set 329 properties, created 329 relationships, completed after 82 ms.)
MATCH (e1:Event) -->(c:Case) <-(e2:Event)
WHERE e2.caseindex - e1.caseindex = 1
CREATE (e2) -[:DF {timebetween: duration.between(e1.completetime, e2.starttime)}]-> (e1)

//5 create offers, relationships offer -> case (Added 25 labels, created 25 nodes, set 150 properties, created 25 relationships, completed after 32 ms.)
LOAD CSV WITH HEADERS FROM "file:///loan_sample.csv" as line
WITH line.case as case,
    line.event as activity,
    line.RequestedAmount as requestedAmount,
    line.FirstWithdrawalAmount as firstAmount,
    line.NumberOfTerms as n_Terms,
    line.OfferID as offer,
    line.resource as resource,
    line.MonthlyCost as monthlyCost,
    line.CreditScore as creditScore,
    line.OfferedAmount as offeredAmount
MATCH (c:Case {name: case}) , (r:Resource {name: resource})
WHERE activity = "O_Created"
CREATE (o:Offer {name: offer, firstamount: firstAmount, n_terms: n_Terms, monthlycost: monthlyCost, offeredamount: offeredAmount, creditscore: creditScore })
CREATE (o) -[:OFFER_TO_CASE]-> (c)

//6 create relationships offer -> event to associate (Offer)events with offers (Created 113 relationships, completed after 19 ms.)
MATCH (e:Event), (o:Offer)
WHERE o.name = e.offerid AND e.offerindex > 0
CREATE (e) -[:EVENT_TO_OFFER]-> (o)

//7 create application nodes (Added 20 labels, created 20 nodes, set 80 properties, completed after 2 ms.)
MATCH (c:Case)
CREATE (a:Application {name: c.name, loangoal: c.loangoal, applicationtype: c.applicationtype, requestedamount: c.requestedamount})

//8 create relationships application -> case (Created 20 relationships, completed after 3 ms.)
MATCH (c:Case), (a:Application)
WHERE c.name = a.name
CREATE (a) -[:APPLICATION_TO_CASE]-> (c)

//9 create relationships application -> event to associate (application)events with application (Created 156 relationships, completed after 5 ms.)
MATCH (e:Event) -[:APPLICATION_TO_CASE]-> (a:Application)
WHERE e.class = "Application"
CREATE (e) -[:EVENT_TO_APPLICATION]-> (a)

//10 create workflow nodes (Added 20 labels, created 20 nodes, set 20 properties, completed after 2 ms.)
MATCH (c:Case)
CREATE (w:Workflow {name: c.name})

//11 relationships workflow -> case (Created 20 relationships, completed after 2 ms.)
MATCH (c:Case), (w:Workflow)
```
WHERE c.name = w.name
CREATE (w) -[:WORKFLOW_TO_CASE]-> (c)
//12 create relationships workflow -> event to associate (workflow)events with workflow (Created 80 relationships , completed after 4 ms.)
MATCH (e: Event) -[:EVENT_TO_CASE]-> (c: Case) <-[:WORKFLOW_TO_CASE]-(w: Workflow)
WHERE e.class = "Workflow"
CREATE (e) -[:EVENT_TO_WORKFLOW]-> (w)
//13 create event chain of offers (Set 88 properties , created 88 relationships , completed after 5 ms.)
MATCH (e1: Event) -[:EVENT_TO_OFFER]-> (o: Offer) <-[:EVENT_TO_OFFER]-(e2: Event)
WHERE e2.offerindex = e1.offerindex = 1
CREATE (e2) -[:O_DF {timebetween: duration.between(e1.completetime, e2.starttime)}]-> (e1)
//14 create application based directly follows relationships (Set 136 properties , created 136 relationships , completed after 15 ms.)
MATCH (e1: Event) -[:EVENT_TO_CASE]-> (c: Case) <-[:EVENT_TO_CASE]-(e2: Event)
WHERE e2.applicationindex = e1.applicationindex = 1 AND e1.applicationindex > 0
CREATE (e1) <-[:A_DF {timebetween: duration.between(e1.completetime, e2.starttime)}]-> (e2)
//15 create workflow based directly follows relationships (Set 60 properties , created 60 relationships , completed after 7 ms.)
MATCH (e1: Event) -[:EVENT_TO_CASE]-> (c: Case) <-[:EVENT_TO_CASE]-(e2: Event)
WHERE e2.workflowindex = e1.workflowindex = 1 AND e1.workflowindex > 0
CREATE (e1) <-[:W_DF {timebetween: duration.between(e1.completetime, e2.starttime)}]-> (e2)
//16 create case-based handover of work relationships (Created 132 relationships , completed after 47 ms.)
MATCH (r1: Resource) -[:RESOURCE_TO_EVENT]-> (e1: Event) <-[:DF]-> (e2: Event) <-[:RESOURCE_TO_EVENT]-> (r2: Resource)
MERGE (r1) -[:HOW]-> (r2)
//17 create offer-based handover of work relationships (Created 55 relationships , completed after 7 ms.)
MATCH (r1: Resource) -[:RESOURCE_TO_EVENT]-> (e1: Event) <-[:O_DF]-(e2: Event) <-[:RESOURCE_TO_EVENT]-> (r2: Resource)
MERGE (r1) -[:O_HOW]-> (r2)
//18 create application-based handover of work relationships (Created 95 relationships , completed after 11 ms.)
MATCH (r1: Resource) -[:RESOURCE_TO_EVENT]-> (e1: Event) <-[:A_DF]-(e2: Event) <-[:RESOURCE_TO_EVENT]-> (r2: Resource)
MERGE (r1) -[:A_HOW]-> (r2)
//19 create workflow-based handover of work relationships (Created 58 relationships , completed after 7 ms.)
MATCH (r1: Resource) -[:RESOURCE_TO_EVENT]-> (e1: Event) <-[:W_DF]-(e2: Event) <-[:RESOURCE_TO_EVENT]-> (r2: Resource)
MERGE (r1) -[:W_HOW]-> (r2)
//20 clean up temporary data (Set 602 properties , completed after 78 ms.)
MATCH (e: Event)
REMOVE e.offerid, e.offerindex

B.2 Full Graph

CREATE UNIQUE CONSTRAINT ON (e: Event) ASSERT e.name IS UNIQUE;
Using Graph Data Structures for Event Logs

CREATE CONSTRAINT ON (c:Case) ASSERT c.name IS UNIQUE;
CREATE CONSTRAINT ON (r:Resource) ASSERT r.name IS UNIQUE;

// 1 create case nodes with case attributes (Added 31509 labels, created 31509 nodes, set 157545 properties, completed after 15973 ms.)

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///loan_full.csv" as line
WITH line.case as case,
    line.LoanGoal as loanGoal,
    line.ApplicationType as applicationType,
    line-RequestedAmount as requestedAmount,
    line.FirstWithdrawalAmount as firstAmount,
    line.NumberOfTerms as n_Terms,
    line.variant_index as vIndex
MERGE (c:Case {name: case, variant: vIndex, loangoal: loanGoal, applicationtype: applicationType, requestedamount: toInteger(requestedAmount)})

// 2 create event nodes with event attributes and relationships to cases (Added 561671 labels, created 561671 nodes, set 6029756 properties, created 561671 relationships, completed after 38875 ms.)

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///loan_full.csv" as line
WITH line.case as case,
    line.EventID as event,
    line.EventOrigin as eventClass,
    line.event as activity,
    line.Action as action,
    line.startTime as startTime,
    line.completeTime as completeTime,
    line.OfferID as offer,
    line.case_index as cIndex,
    line.offer_index as oIndex,
    line.application_index as aIndex,
    line.workflow_index as wIndex,
    line.sY as sY, line.sD as sD, line.sM as sM, line.sHH as sHH, line.sMM as sMM,
    line.sSS as sSS, line.sMS as sMS,
    line.cY as cY, line.cD as cD, line.cM as cM, line.cHH as cHH, line.cMM as cMM,
    line.cSS as cSS, line.cMS as cMS
MATCH (c:Case {name: case})
CREATE (e:Event {name: event, startTime: localdatetime({year: toInteger(sY), month: toInteger(sM), day: toInteger(sD), hour: toInteger(sHH), minute: toInteger(sMM), second: toInteger(sSS), microsecond: toInteger(sMS)}),
            completeTime: localdatetime({year: toInteger(cY), month: toInteger(cM), day: toInteger(cD), hour: toInteger(cHH), minute: toInteger(cMM), second: toInteger(cSS), microsecond: toInteger(cMS)}),
            activity: activity, class: eventClass, action: action, offerid: offer, caseindex: toInteger(cIndex), offerindex: toInteger(oIndex), applicationindex: toInteger(aIndex), workflowindex: toInteger(wIndex)})
CREATE (e) -[:EVENT_TO_CASE]-> (c)

// 3 create nodes for each resource and edges to event nodes (Added 145 labels, created 145 nodes, set 145 properties, created 561671 relationships, completed after 172200 ms.)

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///loan_full.csv" as line
WITH line.resource as resource, line.EventID as event
MATCH (e:Event {name: event})
MERGE (r:Resource {name: resource})
CREATE (r) -[:RESOURCE_TO_EVENT]-> (e)

// 4 create directly follows relationships for events (Set 530162 properties, created 530162 relationships, completed after 22645 ms.)
MATCH (e1: Event) −> (c: Case) −<− (e2: Event)
WHERE e2.caseindex = e1.caseindex = 1
CREATE (e2) −[:DF { timebetween: duration.between(e1.compleatetime, e2.starttime) }]−> (e1)

//5 create offers, relationships offer → case  (Added 42995 labels, created 42995
nodes, set 257970 properties, created 42995 relationships, completed after 10830
ms.)

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///loan_full.csv" as line
WITH line.case as case,
    line.event as activity,
    lineRequestedAmount as requestedAmount,
    line.FirstWithdrawalAmount as firstAmount,
    line.NumberOfTerms as n_Terms,
    line.OfferID as offer,
    line.resource as resource,
    line.MonthlyCost as monthlyCost,
    line.CreditScore as creditScore,
    line.OfferedAmount as offeredAmount
MATCH (c: Case {name: case}), (r: Resource {name: resource})
WHERE activity = "O_Created"
CREATE (o: Offer {name: offer, firstAmount: firstAmount, n_Terms: n_Terms,
    monthlyCost: monthlyCost, offeredAmount: offeredAmount, creditScore: creditScore
})
CREATE (o) −[:OFFER_TO_CASE]−> (c)

//6 create relationships offer → event to associate (Offer)events with offers
(Created 193849 relationships, completed after 1299 ms.)
MATCH (e: Event), (o: Offer)
WHERE o.name = e.offerid AND e.offerindex > 0
CREATE (c) −[:EVENT_TO_OFFER]−> (o)

//7 create application nodes, (Added 31509 labels, created 31509 nodes, set 126036
properties, completed after 298 ms.)
MATCH (c: Case)
CREATE (a: Application {name: c.name, loangoal: c.loangoal, applicationtype: c.
    applicationtype, requestedamount: c.requestedamount})

//8 create relationships application → case (Created 31509 relationships,
    completed after 423 ms.)
MATCH (c: Case), (a: Application)
WHERE c.name = a.name
CREATE (a) −[:APPLICATION_TO_CASE]−> (c)

//9 create relationships application → event to associate (application)events with
    application (Created 239595 relationships, completed after 1209 ms.)
MATCH (e: Event) −[:EVENT_TO_CASE]−> (c: Case) −[:APPLICATION_TO_CASE]−> (a: Application)
WHERE e.class = "Application"
CREATE (e) −[:EVENT_TO_APPLICATION]−> (a)

//10 create workflow nodes, (Added 31509 labels, created 31509 nodes, set 31509
properties, completed after 152 ms.)
MATCH (c: Case)
CREATE (w: Workflow {name: c.name})

//11 relationships workflow → case(Created 31509 relationships, completed after
    383 ms.)
MATCH (c: Case), (w: Workflow)
WHERE c.name = w.name
CREATE (w) −[:WORKFLOW_TO_CASE]−> (c)
101 /\ 12 create relationships workflow -> event to associate (workflow) events with workflow (Created 128227 relationships, completed after 813 ms.)
102 MATCH (e:Event) -[:EVENT_TO_CASE]-(c:Case) -[:WORKFLOW_TO_CASE]-(w:Workflow)
103 WHERE e.class = "Workflow"
104 CREATE (c) -[:EVENT_TO_WORKFLOW]-(w)

106 /\ 13 create event chain of offers (Set 150854 properties, created 150854 relationships, completed after 2248 ms.)
107 MATCH (e1:Event) -[:EVENT_TO_OFFER]-(o:Offer) -[:EVENT_TO_OFFER]-(e2:Event)
108 WHERE e2.offerindex - e1.offerindex = 1
109 CREATE (e2) -[:O_DF{timebetween: duration.between(e1.completetime, e2.starttime)}]-(e1)

112 /\ 14 create application based directly follows relationships (Set 208086 properties, created 208086 relationships, completed after 10294 ms.)
113 MATCH (e1:Event) -[:EVENT_TO_CASE]-(c:Case) -[:EVENT_TO_CASE]-(e2:Event)
114 WHERE e2.applicationindex - e1.applicationindex = 1 AND e1.applicationindex > 0
115 CREATE (e1) -[:A_DF{timebetween: duration.between(e1.completetime, e2.starttime)}]-(e2)

117 /\ 15 create workflow based directly follows relationships (Set 96727 properties, created 96727 relationships, completed after 5218 ms.)
118 MATCH (e1:Event) -[:EVENT_TO_CASE]-(c:Case) -[:EVENT_TO_CASE]-(e2:Event)
119 WHERE e2.workflowindex - e1.workflowindex = 1 AND e1.workflowindex > 0
120 CREATE (e1) -[:W_DF{timebetween: duration.between(e1.completetime, e2.starttime)}]-(e2)

126 /\ 16 create case-based handover of work relationships (Created 11181 relationships, completed after 15588 ms.)
127 MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-(e1:Event) -[:DF]-(e2:Event) -[:RESOURCE_TO_EVENT]-(r2:Resource)
128 MERGE (r1) -[:HOW]-(r2)

132 /\ 17 create offer-based handover of work relationships (Created 6313 relationships, completed after 3739 ms.)
133 MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-(e1:Event) -[:O_DF]-(e2:Event) -[:RESOURCE_TO_EVENT]-(r2:Resource)
134 MERGE (r1) -[:O_HOW]-(r2)

138 /\ 18 create application-based handover of work relationships (Created 8205 relationships, completed after 5716 ms.)
139 MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-(e1:Event) -[:A_DF]-(e2:Event) -[:RESOURCE_TO_EVENT]-(r2:Resource)
140 MERGE (r1) -[:A_HOW]-(r2)

145 /\ 19 create workflow-based handover of work relationships (Created 9353 relationships, completed after 3591 ms.)
146 MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-(e1:Event) -[:W_DF]-(e2:Event) -[:RESOURCE_TO_EVENT]-(r2:Resource)
147 MERGE (r1) -[:W_HOW]-(r2)

150 /\ 20 clean up temporary data (Set 974717 properties, completed after 1394 ms.)
151 MATCH (e:Event)
152 REMOVE e.offerid, e.offerindex
C Screenshots

This appendix includes all screenshots and python scripts for the queries in chapter 5. Picture (a) is always the Neo4j screenshot and picture (b) is always a screenshot of a tool used with the sequential log.

C.1 Screenshots Questions 1 - 11

Figure C.1: Question 1 Screenshots

Figure C.3: Question 3 Screenshots
Using Graph Data Structures for Event Logs

Figure C.2: Question 2 Screenshots

Figure C.4: Question 4 Screenshots
Figure C.5: Question 5 Screenshots
Figure C.6: Question 6 Screenshots
(a)

(b)

Figure C.7: Question 7 Screenshots
$ MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-> (e1:Event) <-[::DF]-

<table>
<thead>
<tr>
<th>&quot;from&quot;</th>
<th>&quot;to&quot;</th>
<th>&quot;frequency&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;name&quot;:&quot;User_1&quot;}</td>
<td>{&quot;name&quot;:&quot;User_1&quot;}</td>
<td>40</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_28&quot;}</td>
<td>{&quot;name&quot;:&quot;User_28&quot;}</td>
<td>14</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_49&quot;}</td>
<td>{&quot;name&quot;:&quot;User_49&quot;}</td>
<td>12</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_61&quot;}</td>
<td>{&quot;name&quot;:&quot;User_61&quot;}</td>
<td>11</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_78&quot;}</td>
<td>{&quot;name&quot;:&quot;User_78&quot;}</td>
<td>9</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_121&quot;}</td>
<td>{&quot;name&quot;:&quot;User_121&quot;}</td>
<td>8</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_40&quot;}</td>
<td>{&quot;name&quot;:&quot;User_40&quot;}</td>
<td>8</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_75&quot;}</td>
<td>{&quot;name&quot;:&quot;User_75&quot;}</td>
<td>8</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_5&quot;}</td>
<td>{&quot;name&quot;:&quot;User_5&quot;}</td>
<td>7</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_7&quot;}</td>
<td>{&quot;name&quot;:&quot;User_7&quot;}</td>
<td>6</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_123&quot;}</td>
<td>{&quot;name&quot;:&quot;User_123&quot;}</td>
<td>6</td>
</tr>
<tr>
<td>{&quot;name&quot;:&quot;User_24&quot;}</td>
<td>{&quot;name&quot;:&quot;User_24&quot;}</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure C.8: Question 7 Frequency Screenshot
Figure C.9: Question 8 Screenshots
$$\text{MATCH (r1:Resource) -[:RESOURCE_TO_EVENT]-> (e1:Event) <-[:O_DF]-}$$

<table>
<thead>
<tr>
<th>&quot;from&quot;</th>
<th>&quot;to&quot;</th>
<th>&quot;frequency&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;User_28&quot;</td>
<td>&quot;User_28&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;User_61&quot;</td>
<td>&quot;User_61&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;User_5&quot;</td>
<td>&quot;User_5&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;User_78&quot;</td>
<td>&quot;User_78&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;User_49&quot;</td>
<td>&quot;User_49&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;User_17&quot;</td>
<td>&quot;User_17&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;User_21&quot;</td>
<td>&quot;User_21&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;User_78&quot;</td>
<td>&quot;User_1&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;User_7&quot;</td>
<td>&quot;User_7&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;User_85&quot;</td>
<td>&quot;User_85&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;User_56&quot;</td>
<td>&quot;User_56&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;User_52&quot;</td>
<td>&quot;User_52&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure C.10: Question 8 Frequency Screenshot
Using Graph Data Structures for Event Logs

(a)

```
{ "activity": "C_Create", "applicationnumber": 6, "scheitzukses": 1, "completedtime": "2016-11-12T04:13:068000000", "statechange": "Initial", "classification": "offer", "eventtime": "2016-10-20T04:48.383000000", "class": "offer" }
```

(b)

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Events</th>
<th>Variant</th>
<th>Started</th>
<th>Finished</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offer_214039778</td>
<td>8</td>
<td>Varia...</td>
<td>28.10.2016 17:58:48</td>
<td>22.11.2016 08:42:12</td>
<td>24 days, 16 hours</td>
</tr>
<tr>
<td>Offer_713767990</td>
<td>7</td>
<td>Varia...</td>
<td>25.10.2016 10:34:13</td>
<td>11.11.2016 08:33:33</td>
<td>16 days, 22 hours</td>
</tr>
<tr>
<td>Offer_116103384</td>
<td>8</td>
<td>Varia...</td>
<td>02.05.2016 11:00:01</td>
<td>18.05.2016 14:11:06</td>
<td>16 days, 3 hours</td>
</tr>
<tr>
<td>Offer_14161171205</td>
<td>7</td>
<td>Varia...</td>
<td>21.07.2016 14:43:04</td>
<td>03.08.2016 16:44:55</td>
<td>13 days, 2 hours</td>
</tr>
<tr>
<td>Offer_217779229</td>
<td>5</td>
<td>Varia...</td>
<td>30.06.2016 15:30:21</td>
<td>13.07.2016 09:43:32</td>
<td>12 days, 18 hours</td>
</tr>
<tr>
<td>Offer_228909160</td>
<td>6</td>
<td>Varia...</td>
<td>16.11.2016 10:16:06</td>
<td>28.11.2016 12:24:51</td>
<td>11 days, 17 hours</td>
</tr>
<tr>
<td>Offer_658074264</td>
<td>5</td>
<td>Varia...</td>
<td>06.08.2016 16:48:14</td>
<td>20.06.2016 09:22:41</td>
<td>11 days, 16 hours</td>
</tr>
<tr>
<td>Offer_473389868</td>
<td>7</td>
<td>Varia...</td>
<td>17.06.2016 14:56:23</td>
<td>28.06.2016 14:23:02</td>
<td>10 days, 23 hours</td>
</tr>
<tr>
<td>Offer_202484808</td>
<td>6</td>
<td>Varia...</td>
<td>27.07.2016 10:12:34</td>
<td>06.08.2016 09:18:51</td>
<td>8 days, 23 hours</td>
</tr>
<tr>
<td>Offer_110013904</td>
<td>7</td>
<td>Varia...</td>
<td>22.04.2016 16:40:36</td>
<td>29.04.2016 16:08:15</td>
<td>6 days, 23 hours</td>
</tr>
<tr>
<td>Offer_1822741376</td>
<td>6</td>
<td>Varia...</td>
<td>24.03.2016 19:11:53</td>
<td>31.03.2016 17:13:46</td>
<td>7 days, 21 hours</td>
</tr>
<tr>
<td>Offer_104587656</td>
<td>6</td>
<td>Varia...</td>
<td>17.03.2016 16:10:41</td>
<td>24.03.2016 10:11:04</td>
<td>10 days, 11 hours</td>
</tr>
<tr>
<td>Offer_855278225</td>
<td>4</td>
<td>Varia...</td>
<td>10.11.2016 10:06:38</td>
<td>10.11.2016 14:48:07</td>
<td>4 hours, 41 mins</td>
</tr>
<tr>
<td>Offer_569323506</td>
<td>3</td>
<td>Varia...</td>
<td>40.04.2016 17:02:38</td>
<td>40.04.2016 17:02:38</td>
<td>0 days, 0 hours</td>
</tr>
</tbody>
</table>

Figure C.11: Question 9 Screenshots
Figure C.12: Question 10 Screenshots
C.2 Question 12

C.2.1 Python Script

```python
import pandas as pd
from py2neo import Graph
import time

loan_raw = pd.read_csv('bpiChallenge17.csv', keep_default_na=False)  # load full log
from csv
loan_raw.drop_duplicates(keep='first', inplace=True)  # remove duplicates from the dataset
loan_raw = loan_raw.reset_index(drop=True)  # renew the index to close gaps of removed duplicates

file = 'BPIC17_clean.csv'
loan_raw.to_csv(file, index=False)
```
```python
sampleIds = loan_raw.case.unique().tolist() # create a list of all cases in the dataset
noOfCases = len(cases)

start = time.time() ### start csv analysis
print('Start CSV analysis...')

log = []

# for id in sampleIds:
for case in sampleIds:
    offerIds = [] # list to keep track of offerIDs within a case
    offerSequences = [] #list of activity-sequences for offers to the corresponding offerIds variable: offerIds[0] refers to offerSequences[0], etc.
    caseSequence = [] # list of activity-sequences for the case
    i = 0 #variable used for temp indices
    j = 0 #variable used for temp indices
    matchCount = 0 # count the "hits" -- i.e. a hit = one offer where O_Created is directly followed by O_Cancelled of the same offer.

    for index, row in loan_raw[loan_raw.case == case].iterrows():
        rowList = list(row)
        if (row['EventOrigin']=='Offer' and row['event'] != "O_Create Offer"): # only execute if event is an offer event except O_Create Offer
            if (row['OfferID'] in offerIds): #if offerID has been observed before
                i = offerIds.index(row['OfferID']) #get index of that offerID
                offerSequences[i].append(row['event']) #add the activity to the sequence of the offer
            else:
                offerIds.append(row['OfferID']) #add offerID to id list
                i = offerIds.index(row['OfferID']) #get index of that offerID
                offerSequences.append([row['event']]) #add the activity to the sequence of the offer
            caseSequence.append(rowList) #add the complete event information (row) to case sequence list

        offerMatches = [False for i in range(len(offerIds))]
        for sequence in offerSequences:
            for activity in sequence:
                if (activity == 'O_Cancelled' and len(sequence) > 1): #if current event is not the first of the sequence and activity is = O_Cancelled
                    i = sequence.index(activity) #get the current activity index in the sequence list
                    if (sequence[i-1] == "O_Created"): #if the preceding activity
                        matchCount += 1 #count as hit
                        offerMatches[j] = True #mark as match
                        j += 1

        if (matchCount > 1):
            log.append([caseSequence, offerIds, offerSequences, offerMatches]) #add case, offerIds+sequences+boolean if an offer is a "match" to new "log" if case has at least 2 hits

header = ['CaseSequence', 'OfferIds', 'OfferSequences', 'OfferMatches']
LogDf = pd.DataFrame(log, columns=header) #create pandas dataframe
#now we have identified all cases that meet the requirements of >= 2 offers with O_Created directly followed by O_Cancelled (with ALL their offers)

finalLog = [] #will contain a list for each offer with caseID, offerID and the sequence from start to the case till O_Cancelled of that offer
for i in LogDf.index: #for every object in the data frame
    caseId = LogDf.iloc[i]['CaseSequence'][0][0] #get caseID
    for offerId in LogDf.iloc[i]['OfferIds']:
        for index in LogDfiloc[i]['OfferIds'].index(offerId)
```
Using Graph Data Structures for Event Logs

```python
# the "target" sequence per offer (A_Create Application to O_Created)
tSequence = []

if (LogDf.iloc[i]['OfferMatches'][index] == True): # only do for matching offers
    for activity in LogDf.iloc[i]['CaseSequence']:
        tSequence.append(activity[1]) # keep adding activities to that offers' sequence

    break

finalLog.append([caseId, offerId, tSequence]) # add that offers' sequence to the final log

header = ['CaseID', 'OfferID', 'OfferSequence']
LogDfCSV = pd.DataFrame(finalLog, columns=header) # create pandas dataframe (for stats only)
print(str(len(LogDfCSV.CaseID.unique())) + ' Cases with ' + str(len(LogDfCSV)) + ' Offers have been identified.')
end = time.time() # end csv analysis
print("The analysis of the CSV file took " + str((end - start)) + " seconds to complete.

start = time.time() # start graph analysis
print('Start Graph analysis...'

graph = Graph(password="1234") # connect to local Neo4j DB with password

# define the query
query = """
MATCH (c1:Event {activity: "O_Created"})::<[:O_DF]-(c2:Event {activity: "O_Cancelled"})::<[:EVENT_TO_OFFER]-(o:Offer)::<[:OFFER_TO_CASE]-(c:Case)
WITH c AS c, count(o) AS ct
WHERE ct > 1
MATCH (:Event {activity: "O_Created"})::<[:O_DF]-(e:Event {activity: "O_Cancelled"})::<[:EVENT_TO_OFFER]-(o:Offer)::<[:OFFER_TO_CASE]-(c)
WITH c AS c, o AS o, e AS O_Cancelled
MATCH p = (A_Created:Event {activity: "A_Create Application"}) <-[:DF*]-(O_Cancelled:Event {activity: "O_Cancelled"}), (O_Cancelled) <-[:EVENT_TO_CASE]-(c)
RETURN p.c
""

data = graph.run(query) # run query on Neo4j DB

output = pd.DataFrame(list(x) for x in data) # save the output to pandas dataframe
finalLogGraph = [] #
for index, p in output.iterrows():
    path = []
    offerId = output[0][index].nodes[-1]['offerId'] # get offerID from the current traces' last node (every trace consists of activities from A_Create Application to O_Cancelled)
    caseId = output[1][index]['name'] # get caseID, match/return of the case in the query has been included to verify the case->offer->activity sequence
    for node in output[0][index].nodes: # walk over the nodes
        path.append(node['activity']) # and add the activities in their order to a list
    finalLogGraph.append([caseId, offerId, path]) # add caseId, offerId and sequence to list to have the data in the same output data format as the csv analysis

header = ['CaseID', 'OfferID', 'OfferSequence']
LogDFG = pd.DataFrame(finalLogGraph, columns=header) # create pandas dataframe (for stats only)
```
Using Graph Data Structures for Event Logs

```python
print(str(len(LogDfG['CaseID'].unique())) + ' Cases with ' + str(len(LogDfG)) + ' Offers have been identified."

end = time.time()  # end Graph analysis

print("The analysis of the graph query took " + str((end - start)) + " seconds to complete."

print('Comparing results...')
if (sorted(finalLog) == sorted(finalLogGraph)):  # check if (sorted list) results are equal
    print('The results match')
else:
    print('The results do NOT match')
```

C.2.2 Output

Start CSV analysis...
103 Cases with 218 Offers have been identified.
The analysis of the CSV file took 927.1300814151764 seconds to complete.
Start Graph analysis...
103 Cases with 218 Offers have been identified.
The analysis of the graph query took 0.0635610580444336 seconds to complete.
Comparing results...
The results match

Figure C.14: Output of the Q12 Python Script

C.2.3 Execution Times on Full Graph

<table>
<thead>
<tr>
<th>Question</th>
<th>Execution Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>case</td>
</tr>
<tr>
<td>2</td>
<td>event</td>
</tr>
<tr>
<td>3</td>
<td>startTime</td>
</tr>
<tr>
<td>4</td>
<td>completeTime</td>
</tr>
<tr>
<td>5</td>
<td>LoanGoal</td>
</tr>
<tr>
<td>6</td>
<td>ApplicationType</td>
</tr>
<tr>
<td>7</td>
<td>RequestedAmount</td>
</tr>
<tr>
<td>8</td>
<td>Accepted</td>
</tr>
<tr>
<td>9</td>
<td>Action</td>
</tr>
<tr>
<td>10</td>
<td>FirstWithdrawalAmount</td>
</tr>
<tr>
<td>11</td>
<td>NumberOfTerms</td>
</tr>
<tr>
<td>12</td>
<td>OfferID</td>
</tr>
</tbody>
</table>

Table C.1: Query Execution Times
Bibliography


