Eindhoven University of Technology

MASTER

A schema framework for graph event data

Esser, S.

Award date:
2020

Link to publication

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

A Schema Framework for Graph Event Data

Master Thesis

Stefan Esser

Supervisors:
dr.   Dirk Fahland
dr.   George Fletcher
dr.   Oktay Türetken

Final version

Eindhoven, February 2020

Abstract

Process event data contains events related to various entities of a process. These multi-dimensional event data, generated by information systems, serve as source for the retrieval of event logs for process mining techniques. The sequential event logs used in process mining today are bound to a single case notion for their events to relate. This means that we are forced to flatten the data into a single-dimensional format by dropping/ignoring information about further related entities when creating event logs. Many processes cannot be fully captured by such a data structure. Even though relational databases can be used to store these multi-dimensional event data, but existing query languages do not support querying for paths of temporal relations of events, especially across multiple entities. Property graphs support storing and querying such data, but there is no common understanding of how event data is systematically encoded in property graphs and how such a data model can formally be defined such that structural and behavioral rules on the data can be governed. Because no standard property graph schema definition exists, we first introduce a schema language to describe the global data structure of graph databases with event data. We furthermore define property graph-based templates to encode the fundamental event log concepts. To enable the definition of integrity constraints, i.e. structural and behavioral rules, on instance level, we introduce a technique that is based on local patterns (sub-graphs) of the global schema to define rules within that limited scope. A set of templates for such local patterns has been developed and successfully tested in six case studies on five different event logs data sets. Furthermore, we provide basic means to validate the compliance of a graph instance with event data to such a schema structure. Python and Cypher templates have been developed for the creation of the six different graph event logs. These graph event logs and their creation script templates are the direct results of our case studies and thus can serve as baseline for new research question that arose from the different design artifacts of this thesis.

Keywords: process mining, multi-dimensional event data, labeled property graph, database schema, graph database

Preface

This master thesis concludes my graduation project for the Master's program in Business Information Systems at Eindhoven Univertiy of Technology (TU/e), conducted within the Architecture of Information Systems (AIS) research group at the Department of Mathematics & Computer Science.

First and foremost, I want to thank Kathrin and the rest of my family for giving me unfailing support throughout the entire period of my studies. You helped me keeping my goals in sight and finally - with this thesis - reaching them.

Second, I want to thank my graduation supervisor Dirk Fahland for his great support, guidance and for all the (free-)time you spent on my final project. Not only as a supervisor but also as a mentor throughout my entire master program, he always had his door open for me and my fellow students. He managed to spark my interest in science and his attitude to help students to excel themselves is admirable.

Furthermore, I want to thank George Fletcher, my second supervisor, for our regular meetings and inspiring discussions and Oktay Türetken, my third supervisor, for his valuable input.

I also want to thank the AIS group for the nice time while working, discussing and researching together.

Last, but certainly not least, I want to thank my fellow students Eva, Kavya, Ioannis, Miro and Przemek for a great time while studying together.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This master thesis concludes my graduation project for the Master's program in Business Information Systems at Eindhoven Univertiy of Technology (TU/e), conducted within the Architecture of Information Systems (AIS) research group at the Department of Mathematics & Computer Science.

In this chapter, we first provide the overall context of this thesis in section 1.1. Then we describe the research problem and set the goals according to the defined requirements of our research in section 1.2. Section 1.3 introduces the methodology we used to conduct our research.

1.1 Thesis Context

Process mining is the domain that combines data mining with business process management to analyze process event data captured from information systems (IS). Retrieving such (sub-)sets of event data is a recurring activity in process analysis and process mining [27]. For the analysis with process mining techniques, these collections of *events* are stored in *event logs*, where every event refers to a specific *case*, i.e. to a specific instance of a process execution. Typically, one event describes an *activity* of such a process execution and it always contains temporal information, i.e. the activity time, such that we can determine the order of executed activities w.r.t. to a case as shown with the "Timestamp" column in table 1.1. One process execution is stored as a sequence of activities and denoted as *trace*.

For example, the basic example log in table 1.1 contains two cases identified by "cID" with the traces $\langle a, b \rangle$ and $\langle b, a \rangle$. Activity a is followed by b in trace 1 and activity b is followed by a in trace 2. When referring to the individual events with additional attributes, rather than to their activities only, we can create an event log table as shown in table 4.1. This table contains more attributes, but only one trace. Such traces can easily be queried and for behavioral properties such as event sequences and temporal relations like directly and eventually follows between events and can be combined with other data attributes [6, 10, 19, 24, 25, 26], e.g. resource information, i.e. which resource executed what activity.

cID	Activity	Timestamp	...
1	a	29.08.19 10:30	...
1	b	29.08.19 13:14	...
2	b	29.08.19 10:35	...
2	a	30.08.19 13:59	...
...	...	...	...

Table 1.1: Basic Event Log Example

In reality, however, many processes cannot be captured by only a single case identifier as can be seen in the example case of a loan application process in table 4.1 where one loan application is related to two loan offers. This work specifically addresses processes containing multiple case notions, i.e. different types of entities exist in a process that can be related to each other resulting events with various relations to different entities, i.e. multi-dimensional event data. Various approaches on extracting, representing, storing and mining multi-dimensional data [8, 13, 20, 28] have recently been developed, where concepts like business artifacts or process objects have been utilized to model the various multi-dimensional relations of the data. Relational databases (RDBs), for example, can store 1:n and n:m relations of events and case identifiers and between case identifiers. The explicit behavioral information of event sequences with arbitrary length, however, is lost when extracting an event log, because the creator is forced to flatten the event data [17] by choosing a single case notion. Thus, we generate a very narrow view on the process. Reconstructing such sequences with multiple case identifiers from an RDB [17, 20] can only be done under severe loss of information and can only be done with an arbitrary number of joins and by large and non-intuitive queries [9].

Literature shows that different approaches have already been researched to deal with multi-dimensional event data. Business artifacts and artifact-centric process models [21, 7, 20, 23] emphasize interactions between different information objects in a process. Such approaches require different source data structures than traditional event logs. Recent research also looks into extending traditional event logs w.r.t to business objects [28], i.e. information objects of a process such as invoice or an order document, by adding a column per object type (case notion) of a process. These columns contain sets of objects to which one event relates.

With the goal to find a suitable data model to store and query multi-dimensional event data, we explored the feasibility of storing and querying multi-dimensional event data in labeled property graph (LPG) databases in the preceding works of the thesis [11, 12]. We have been able to show that event data and its variety of relations can be stored in LPGs and queries on such graph event data can be formulated by implementing the BPI Challenge 17 data set (BPIC 17)[32] in the graph database system (GDBS) Neo4j. For storing and querying the data, we implicitly respected logical, behavioral rules, e.g. two resources may only handover work to each other if they have relations to two events with a directly follows relation, separately from the structural data model of the graph. For example, in table 1.1, the events of activity *a* and *b* of case *1* are related such that *a* is directly followed by *b* because there is no other event, w.r.t. case *1*, in between them. Consequently, a resource related to *a* can handover work to the resource of *b*. Especially queries over sequences of multiple entities, which are, if at all possible, very hard to define in SQL for RDBs, can be defined and evaluated relatively quick with a few queries in Cypher, Neo4j's native query language. Our previous research, however, has only been conducted on a single event log and lacks a formal description for the data model of such graph event data, i.e. the definition how to systematically encode this event data in LPGs to create graph event data according to some schema. This motivated us to put the focus of this thesis on the development of a systematic approach to encode event data in a property graph and a schema definition for such data.

In property graphs, data points and their relations are treated as "first-class-citizens" in terms of a graph of nodes and edges. As LPGs are so flexible, in principle any structure can be created. This, from a data modeling perspective, is not a good thing as a limited data structure helps accessing and querying the data and makes sure the ideas are expressed the same way. Developing a schema standard for property graphs is ongoing research [4, 5] and intensively discussed in academia and industry. So what is necessary to govern a graph event log? For example, we want to limit the creation of relationships and nodes based on the existence, non-existence or depending on certain property values of other graph elements. We furthermore want to make sure that a certain type of nodes exists, such as event nodes. Generally, we want some formalized means of writing the implicit rules we applied in our preceding case study, i.e. we want to identify a way to govern the creation of structural and behavioral relations in a graph database so that we adequately represent process event log concepts such as directly follows and handover of work.

1.2 Research Problem & Goals

As briefly discussed in section 1.1, sequential event logs with a single case identifier are not sufficient to store multi-dimensional event data as they assume that only a single case notion exists and every event refers to exactly one case. This thesis builds upon our research on storing and querying multi-dimensional event data [11, 12]. In this exploratory work we showed that LPGs can be used to store and query multi-dimensional graph data.

When we started to study the possibilities to define such a schema for labeled property graph schema, we have been confronted with an additional issue, i.e. no standard schema or data definition language exists for LPGs [4, 5].

Problem Statement:
How can we define a schema for property graph for multi-dimensional event data and systematically encode the event data in a property graph instance accordingly, to enable querying of the event data with reliable and repeatable results?

The lack of a standard schema for property graphs, however, does not allow to define a generic data model for graph event data with integrity constraints, yet. Consequently, no standard approach for defining, storing and querying graph event data could be developed. Our research has been conducted with the overall goal of creating design artifacts for a property graph schema with the following requirements:

1. We must be able to describe/define the data model of a labeled property graph containing event data.

2. Constraint definitions and encoding of behavioral attributes to property graphs must be possible in a schema.

3. Encoding of event log concepts in such a graph must be possible.

4. It must be possible to relate different case notions in one log.

We have used these five requirements (R1-4) and the research problem above to set the following goals for the development of the research output of this thesis:

Goal Statement:
Enable the definition of graph event logs based on property graphs described by a formally defined schema with constraints and encoded behavioral attributes.
We divide the goal into four sub goals:

1. Develop a *schema definition for property graphs* with *integrity constraints* to explicitly model event data concepts.

2. Define a baseline for *encoding event data in property graphs.*

3. Provide basic means of *schema verification* for *LPGs.*

4. Develop a generic *concept for process entities* in multi-dimensional event data in LPGs.

Goal 1 (G1) is set to tackle the requirements R1 and R2, as the desired outcome here is to provide means to model a property graph schema that is able to constraint data structures on instance level. With G2 we aim to fulfill R3, as by reaching G2 we will not only show that encoding is possible, but also provide generic translation templates for such encodings. G3 does not tackle G1 and G2 directly, but is a necessity to enable verification whether R1 and R2 are actually met by our solution. The outcome of G4 shall be a more flexible concept than "case" and thus satisfy R4.

1.3 Method

In this thesis, we followed the design science research (DSR) process [15, 22]. The research goal and the requirements for a solution of this work has been derived from our previous works on graph event data [11, 12].
The overall objective for the research, putting event data into a property graph, was given as input by my supervisor.

The requirements for the solution were derived out of an exploratory case study on the BPIC 17 data set [11, 12] which led to the definition of the above research objectives.

The development of our solution is based on an iterative execution and refinement of schema definitions for five different event logs to identify suitable artifacts for the requirements above. We iteratively conducted the following steps for each data set:

- Prepare the source data format to be suitable for GDBS import.

- Import the event data into a GDBS.

- Transform the event data in the GDBS by adding nodes, relationships and properties to model process behavior.

- Formulate queries over the graph event data.

- Evaluate how the importing, transformation and resulting graph data models compare to other data sets.

- Compare how queries on this data are different or similar to queries on other data sets.

- Evaluate how good the graph event data describes the process and domain concepts.

- Evaluate the complexity in terms of import and transformations.

- Compare the complexity of the data model in terms of a global schema is to the data models of other data sets.

Along the iterations we elicited principles for LPG schema modeling and graph event data modeling. We used five different real-life event log data sets and for all event logs together we iterated more than 100 times over the steps above.

This process resulted in the creation of several different artifacts:

1. A graph-based language and visualization to describe a global schema of a LPG over a structure of relationships, nodes and properties for event data.

2. A non-exhaustive set of templates for converting event log concepts into LPG data structures.

3. A graph-based language and visualization to describe local, pattern based schema templates with integrity constraints, called rules. We furthermore defined the following examples:

   (a) "0_core" - Ensures the mandatory event log concepts: events, entities (replacing case), activities and timestamps.

   (b) "1_df" - Models the temporal relations between events.

   (c) "1_e_coincide" - Defines how to correlate coinciding events.

   (d) "1_en_coincide" - Defines how to correlate coinciding entities.

   (e) "2_how" - Models handover of work relations between resources of events.

4. A basic validation method for global schemata and local schemata with rules.

5. The concept of a generic entity, to remove the limitation to one case identifier.

6. A set of Python and Cypher templates to create graph event logs from five different event logs.

7. Six graph event log instances of real-life event data with multiple entity identifiers.

In the remainder of the thesis we first provide necessary background knowledge in Chapter 2. In Chapter 3 we introduce a schema language and visual representation for a global property graph schema, followed by the definition of templates for event log concepts in property graphs in Chapter 3. Chapter 5 introduces a concept of local, pattern-based schema definition with complementary rules to add (local) integrity constraints to the global schema. In Chapter 6 we provide a set of templates for local pattern schemata to demonstrate them in the evaluation with the five data sets in Chapter 7. We conclude this thesis in Chapter 8.

Chapter 2

Background

This chapter introduces existing concepts on process event logs and graph databases this thesis relies on. Section 2.1 introduces detailed information about event data and event logs. In section 2.2 we point out the specifics of multidimensional event data and how traditional event logs are not suitable to fully encode these data. In section 2.3 we describe graph databases in general and then describe the labeled property graph model we used for our research in section 2.4. Furthermore, we introduce Neo4j, a graph database management system (GDBMS) and Cypher, a query language for property graphs [14] in section 2.5. In section 2.6 we recall basic principles of database schemata.

2.1 Event Data and Event Logs

Table 2.1 shows a simple event log [27] with one case (process execution). Each row is one *event*, each column is an *attribute* and each cell is an *attribute value*. Some of these attributes belong to the case, for example the "Amount" column. Other attributes belong to the events such as the *activity* which indicates what process task led to the creation of an event.

Say we are looking at a loan application process of a local bank where customers can apply for private loans and if they meet the banks requirements, the applicants receive one or more offers with different conditions for a loan. An example case of such an event log is shown in table 2.1.

Process-related event data is usually created and updated by some information system by creating data points for activities executed in some process. Single data points of these event data are referred to as events. Typically, each process event log is centered around an entity, often referred to as case, that is updated by the process. For example, the event log in table 2.1 is centered around an application entity. Most organizations, however don't structure their processes around a single entity. For example the "oID" column in table 2.1 indicates relations of certain events to a second entity type: offer.

As described by van der Aalst in Chapter 5 in [27], event logs need to fulfill some minimum requirements in order to be usable for process mining. Requirement 1) is the limitation to one case identifier. One case is basically an instance of a process, i.e. a process execution, and the case identifier is defined according to what entity is subject to the log. Requirement 2) states that each event must have an activity attribute. Requirement 3) is the timestamp or temporal ordering attribute to events. The events in the log are ordered sequentially by their temporal attributes with respect to their case. In other words, the events and their order in the log are determined by the case identifier making the log a view on the process.

To determine the temporal order, events are required to have a timestamp or some other ordering attribute. Keeping the temporal order of events is a very important aspect for the process analysis. Many process mining techniques rely on the directly follows relationships over activities, derived from the directly follows relation of their events, i.e. if event *e1* happens right before event *e2* and there is no event in between within one process instance, then *e2* directly

follows *e1*. Consequently, *e2.Activity* directly follows *e1.Activity*.

In summary, as the very minimum, we need a case identifier and events with activity and ordering attribute for an event log. Further information can be associated to events and cases by event and case attributes respectively. Optionally, further information such as resources can be included. From this basis further concepts can be applied to an event log, such as organizational process mining as described in Chapter 9 of [27].

Traditional event logs for process mining come in a two-dimensional data structure in comma-separated values (CSV) or eXtensible Event Stream (XES) [16] format. While CSV is a simple file format widely known to other domains, XES is a process mining specific data format based on Extensible Markup Language (XML) and the de facto standard format for process mining event logs.

2.2 Multidimensional Event Data

A process execution can involve multiple different entities. For example, the event log in table 2.1 involves applications ("cID") and offers ("oID"). When referring to offer 1, we only consider the two events for the "Create Offer" and "Send Offer". When referring to application, we consider all events shown in table 2.1. By just using the case identifier for process mining analysis, typical techniques see the "Create Offer" and "Send Offer" activities as a repetitions within the case, but these actually belong to two different offer entities.

During a execution of a process with many different entity types, the different entities can interact and thus influence each other. So, which of these entity types shall be a case identifier for a new event log? In fact, every entity type and even any combination of interacting entity types of a process can be used as case identifier.

There we have a 1:n relation between application (*cID*) and offer (*oID*) entities. Other processes have 1:1 and n:m relations between their entities and thus the resulting event data becomes multi-dimensional too.

One major issue of the event logs described in section 2.1 is that creators of these event logs are bound to the single case identifier and thus have to flatten the data from the source IS [17]. In our small example above, we could chose to combine applications and offers to one case to create a log. In the given two-dimensional event log format, however, these multi-dimensional relationships pose a challenge, especially we want to analyse the directly follows relationships of events with respect to different entities.

Previous research in the field of process mining related multi-dimensional event data has been conducted by Lu et al. [20], where business artifacts [21, 7] have been used to mine from multi-dimensional event data. This approach, however, relied on RDB-based event data providing far more expressiveness compared to the event logs we based our research on. Closer to our research is the work of Popova et al. [23] which uses a so called event stream where attributes (columns) can hold multiple values. The recent work of van der Aalst on object-centric process mining [28]

cID	Activity	Timestamp	Amount	oID	Terms
1	Create Appl.	29.08.19 10:30	1000		
1	Appl. Ready	29.08.19 10:35	1000		
1	Create Offer	29.08.19 13:14	1000	1	128
1	Create Offer	29.08.19 13:49	1000	2	256
1	Send Offer	29.08.19 18:00	1000	2	256
1	Send Offer	29.08.19 18:00	1000	1	128
1	Offer Returned	30.08.19 13:49	1000	2	256
1	Appl. Complete	30.08.19 13:59	1000		

Table 2.1: Multi-Dimensional Event Data Case Example

propose a table based format, where relations between events and objects are recorded in a table. Object types are attributes (columns) and the respective object instances are correlated to the events in sets, i.e. a row (event) may have different columns, one for each object type, which indicate what objects (object sets in the field values) may be associated to the event. Besides the question how to store and query these multi-dimensional data recent research is also looking after ways to describe such a complex process behavior [13].

Two problems emphasized in the related works are data divergence and convergence. Data convergence relates one event to multiple cases and data divergence relates multiple executions of the same activities within one case. In RDB source systems, these problems are not very relevant, since the data models can represent the different relationships. In the flattened data of event logs, however, theses relationships pose a challenge. Our research provides a different perspective on these problems, because we take the flattened data from event logs and try to "reverse engineer" the flattening of the data to recreate the lost information. For example, we take the log from table 2.1 and analyze whether its columns may be suitable entity identifiers or not and then define the entities we can derive from it. Sometimes this selection may depend on domain knowledge of the process, but an educated guess can usually be made by a profound analysis of the data.

2.2.1 Entity Concept

In this section we introduce the concept of an entity. The concept in the context of multi-dimensional event data has been largely inspired by the work of Lu et al. [20] and Popova et al. [23] where so-called business artifacts are used in the context of process mining on multi-dimensional data and the work of van der Aalst [28] where objects, a more generic concept than business artifacts, are modeled into the corpus of classical event logs. Entities are individual, information objects of a process that are created, manipulated or somehow involved in a process execution. Entities are more 'flexible' than traditional case identifiers, since any number of them can be correlated to one event and they can even be used to derive entities from attributes seen in the data, e.g. by combining two attributes. The concept of an entity shall replace the traditional case notion, as an entity can be anything from a resource to an application to an offer to any combination (that makes sense from a process perspective) of existing entities. We use entities in place of the traditional case identifier in a graph database such that entities can be used to define further cases from combinations with other entities. The objects in [28] and our entity are very similar, as they both can represent the different case notions of a process. Entities differentiate to these objects such that they also represent information objects of a log that are not necessarily considered a case notion, such as resources. For example, resource information can, with our entity concept, materialize once as case notion such that all related events are in temporal order w.r.t. to the resources (e.g. with "User" = resource, "Patty" has the trace ⟨ "Create Offer", "Send Offer", "Offer Returned" ⟩ in the example in table 4.1), and it can materialize as meta-entity, representing organizational information of a process, i.e. the resource is an attribute to events of other entity types. By defining two different entity types for that same source of information (resource), we can use both in the same graph, once as case and once as resource.

2.3 Graph Databases

A graph database, as the name suggests, relies on mathematical graph data structures. Vertices and edges, or nodes and relationships, as they are commonly referred to in the database domain, build the base of the data models of graph databases. There are two predominant database models in the field, the *Resource Description Framework* (RDF) and the *Labeled Property Graph*. While the RDF world is all about triples, i.e. sets of node-relationship-node triples, to build complex graphs, LPGs give much more freedom to model data structures as any number of properties can be used to add information to nodes and edges. Even though it may sound counter intuitive, compared to relational databases, graph databases put much more emphasis on the relations of data points. As previous research has already shown [12], property graphs can be used to store

and query multi-dimensional event data. At the time of writing, however, there is no standard data definition language or schema for labeled property graphs yet. This made it necessary to further develop the results of [12] and conceptualize event data representation in labeled property graphs. Therefore, the labeled property graph model and the respective schema definition is one of the core aspects of our research.

2.4 Labeled Property Graph

Labeled property graph is the data model we use as basis for our research. LPGs consist of a set of vertices, called nodes, and a set of edges, called relationships. Nodes and relationships can have labels, which effectively group the nodes and relationships into classes or types. Additionally, key-value pairs can be assigned to nodes and relationships. These key-value pairs are called properties. Properties are object-specific, i.e. they are assigned to node and relationship objects and do not depend on the objects label. While there exist LPG based data models of various flavors as described in Chapter 2 of [4], we refer to the property graph model implementation of Neo4j [18]. This means a property graph with directed relationships, allowing multiple relationships between two nodes, zero or any number of labels to nodes, zero or one label to relationships and an arbitrary number of key-value pairs, i.e. properties, to node a node or relationship. The formal definition of the property graph model used in this thesis and the respective visual representation are described in the following two paragraphs.

For this thesis we use the following LPG model definition based on the works of Bonifati et al. [4, 5].

For a formal definition of property graphs assume that

- $\mathcal{O}$ is a set of objects;

- $\mathcal{L}$ is a set of labels;

- $\mathcal{K}$ is a set of property keys;

- $\mathcal{V}$ is a set of values.

We define a property graph as a structure (N, R, η, λ, v) where

- $N \subseteq \mathcal{O}$ is a set of node (vertex) objects;

- $R \subseteq \mathcal{O}$ is a set of relationship (edge) objects;

- N and R are disjoint, $N \cap R = \emptyset$;

- $\eta : R \to N \times N$ is a function assigning to each relationship an ordered pair of nodes;

- $\lambda : N \cup R \to \mathcal{P}(\mathcal{L})$ is a function assigning to each object a finite set of labels such that P(S) denotes the set of finite subsets of set S;

- $v : (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a partial function assigning values for properties to objects.

The basic components of our LPG model are illustrated in figure 2.1.

Nodes filled with colour - colouring only for an optical indication of relationship between node instance - node type.

The nodes are circles filled by different colours. Note that the colours to not imply any semantics, but rather serve the purpose to visually group certain node types (labels). Labels are marked by a leading colon, e.g. *:Student.* The labels of nodes can either be placed inside the node, or, if the space inside the node is to small for the label(s), the labels can be moved to the property box of the node. This property box contains the property keys and values (and if applicable the labels) of the respective object. All relationships are directed, and are represented as arrows originating from the source node and pointing on the destination node. The relationship property
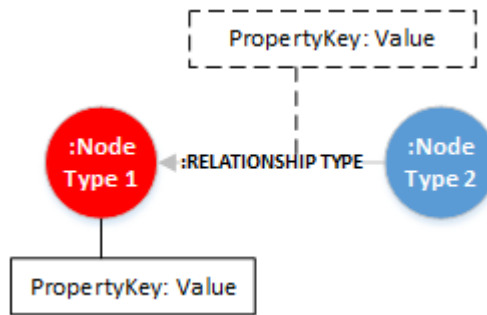
Figure 2.1: Visual Property Graph Instance Elements

boxes are drawn with dashed lines to clearly separate them from the node property boxes which are drawn with solid lines. As standard annotation format we completely capitalize *:RELATIONSHIP* labels and only capitalize the first letter of *:Node* labels to support readability.

Figure 2.2 shows a small example graph.
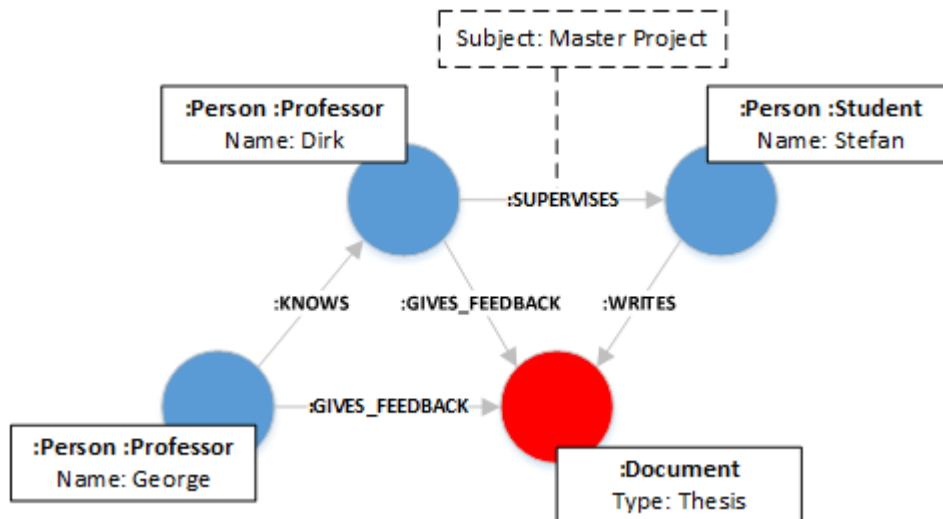
Figure 2.2: Graph Instance Example

The figure shows a property graph that gives some high level context to this thesis. The three blue nodes are persons, two of which are the TU/e professors Dirk and George who helped me, Stefan, with their feedback and supervision to write this thesis. The thesis is a *:Document* node and it has a property "Type: Thesis". Dirk and George of course know each other, which is modeled by the *:KNOWS* relationship between the two nodes. We can also define multiple relationships between nodes, for example Dirk obviously also knows Stefan and he also knows George. Creating a *:KNOWS* relationship from Dirk to George effectively creates a undirected-alike (or bi-directional) *:KNOWS* relationship from by two directed relationships.

2.5 Neo4j and Cypher

The graph implementation for this work has been done based on the GDBMS Neo4j and the graph query language Cypher [14]. As it is crucial to understand the concepts and query structures of Neo4j and Cypher and their importance to the graph and schema concepts we use in this work, we give a short introduction into their components in this section. We first explain how the property graph concepts introduced in section 2.4 are implemented in Cypher followed by the

most important query clauses, operators and functions and how they can be used to query and manipulate LPG instances.

2.5.1 Graph Elements

In the visual appearance for LPG instances we introduced in section 2.4 we already incorporated some of the annotations of Cypher, such as the leading colon for labels (*:Label*). The Neo4j LPG model implements the definitions introduced in section 2.4. Thus, we already provided the basis for the introducing the following concepts.

Nodes

Nodes in Cypher are addressed in round brackets. Empty brackets "()" serve as a wildcard for all nodes in the set of objects we query on. By specifying a label for a node "(:Label)", we restrict the wildcard to all nodes with the label specified in the round brackets. In order to work further with nodes represented by the wildcard, we can add a variable to the node, "(variableName:Label)". The *variableName* can now be used to address the set of nodes with the *:Label*.

Relationships

Relationships follow a similar logic for addressing them. They are addressed by an arrow-like string "−>", implicitly indicating the direction of the relationship. As every relationship is associated with an ordered set of nodes, i.e. a source node (from) and a destination node (to), the Cypher annotation for them always requires two nodes as well, even if no specific nodes are declared. Thus a minimal example to specify a relationship in Cypher is: "()−>()", i.e. a relationship can never be without a source and destination node. The arrow "−>" for directed and the two dashes "−" for undirected relationships serve as wildcard for relationships, similar to the brackets "()" for nodes. Pleas note, that Neo4j (and our property graph definition) only support directed relationships in the data model. Cypher, however, can define queries ignoring the direction of the relationship itself in a pattern. Similar to nodes, the group of relationships can be restricted by specifying a label in squared brackets to the relationship like "()-[:Label]−>()" and again we can address this set of relationships with variables such as "()-[variableName:Label]−>()". Node and relationship variable declarations can freely be combined. With "(n:Professor)-[rel:SUPERVISES]−>()" we declare variable *n* for nodes with label *:Professor* and variable *rel* for relationships with *:SUPERVISES* label. For the example graph in figure 2.2 this pattern would match every professor that supervises someone, or something, because we did not specify a label for the destination node. In the concrete example, *n* would include Dirk's node and *rel* would include the *:SUPERVISES* relationship for Stefan's master project. In bigger graphs, say for the entire TU\e, the variable would contain a lot more nodes and relationships, which already illustrates a core concept of querying graphs: finding sub graphs that match specific patterns.

Properties

Neo4j implements a set of different data types for property values. The most relevant for this work comprise *integer*, *float*, *string*, *boolean*, *list* and *datetime*. Relationships and nodes can hold any number of properties. Properties are specific to an object, i.e. properties and property types are not dependent of a certain label. For example, a relationship and a node can both have a *Name* property and one node with label *:Student* may have a property *StudentNumber* while another *:Student* node doesn't. Properties of nodes or relationships can be addressed in different ways in Cypher. They can, for example be used in the pattern specification directly by extending the node or relationship with the property key and value in curly brackets to further restrict the matched sub graphs. "(n:Person Name: George)" for example will match only those persons with name George. A another way to address properties of objects is to first declare a variable *n* with "(n:Person)" and then, usually in a "WHERE" clause, address the name property of the

nodes in the variable with "n.Name". Along with other Cypher clauses, we elaborate more on the "WHERE" clause in the following section.

2.5.2 Cypher Query Language

In this section we give a detailed introduction into the graph query language Cypher[14].

MATCH

"MATCH" is the clause used to identify the patterns of interest in our graph. It is used with the node, relationship and property notations we introduced in section 2.5.1. In the listing below we show an example query for the "MATCH" clause.

```
1    MATCH (p: Professor) −[:SUPERVISES]−>(s: Person {Name: 'Stefan'})
```
Listing 2.1: MATCH Clause Example

Say we run this query against the small example graph in figure 2.2. The result will be Dirk's node for p and Stefan's node for s.

RETURN

A "MATCH" clause cannot be on it's own, because Cypher always requires some operation on the matched sub graphs. The easiest operation is to simply return the matched sub graphs with a "RETURN" clause as you can see in the listing below.

```
1  MATCH (p: Professor) −[:SUPERVISES]−>(s: Person {Name: 'Stefan'})
2  RETURN p AS ProfessorNodes, s AS StefanNodes
```
Listing 2.2: RETURN Clause Example

The query returns all professors in p that supervise a person with the name Stefan as well as these persons' nodes in s. Actually, there is a little extra to the return statement, because we renamed the output variables with "AS". Thus in the final output, the variable p has the name *ProfessorNodes* and the variable s has the name *StefansNodes* and the original variable names are discarded. Please note that graph objects can multiple time be contained in a variable. Say Dirk, the professor, has two students named Stefan he supervises. This would result in a situation where Dirk's node is matched two times in the "MATCH" clause and thus will be included twice in the variable p (or *ProfessorNodes*), i.e. the variable is a multi set. If this is not intended, we can use the "DISTINCT" clause.

DISTINCT

"DISTINCT" is an aggregation function to ensure that the result contains only distinct objects, i.e. it removes duplicates from a multi set. This means that per object, only one item is included in the variable values. The following listing shows how the example above can be enhanced to include every graph object only once.

```
1  MATCH (p: Professor) −[:SUPERVISES]−>(:Person {Name: 'Stefan'})
2  RETURN DISTINCT (p)
```
Listing 2.3: DISTINCT Clause Example

WHERE

"WHERE" is a clause that helps us to put restrictions on the sample space of our graph. With "WHERE" we can specify requirements for objects in patterns. The following listing shows a short example that combines the aggregation function "COUNT()" with the "WHERE" clause such that the first part of the query determines what professors p will be returned, i.e. only professors that have more than one supervision are part of p.

```
1 MATCH (p:Professor)−[r:SUPERVISES]−>(:Person {Name: 'Stefan'})
2 WITH COUNT(r) AS NoOfSupervisions, p
3 WHERE  NoOfSupervisions > 1 and p.Name = "Dirk"
4 RETURN (p)
```
Listing 2.4: WHERE Clause Example

The listing also gives a good example on how to address properties from variables, since the "WHERE" statement has two components connected by a logical "and". The second condition (p.Name = "Dirk") limits our result to professors named Dirk that have more than one supervision.

WITH

The "WITH" clause can be used to handover results of one query to the next query. If we for example want to execute aggregation opertaions to a variable and hand the aggregated value over to the next query clause, we can use "WITH" to make a cut. After a "WITH" clause, all other information of the previous query statements are lost.

```
1 MATCH (p:Professor)−[rel:SUPERVISES]−>(s:Person {Name: 'Stefan'})
2 WITH COUNT(rel) AS NoOfSupervisions, p
3 RETURN p, NoOfSupervisions
```
Listing 2.5: WITH Clause Example

The COUNT() function aggregates all relationships in *rel* per professor, providing the number of supervised people. The variable *s* is no longer available as output after the "WITH" statement.

ORDER BY

"ORDER BY" is a statement that, for readers familiar with SQL, should be self-explanatory. "ORDER BY" defines the property types after which values the output shall be orderes.

```
1 MATCH (p:Professor)−[r:SUPERVISES]−>(:Person {Name: 'Stefan'})
2 RETURN (p), COUNT(r) AS NoOfSupervisions
3 ORDER BY NoOfSupervisions DESC
```
Listing 2.6: ORDER BY Clause Example

In the listing above, the professors will be returned in the order of their number of supervisions. The "DESC" option, as opposed to "ASC", orders the result in descending order, i.e. the professor with the most supervisions will be on top of the list.

LIMIT

By adding the "LIMIT" keyword to the above example, we would only return the professor with the single most supervisions. Please note that if there are more than one, a random professor with the highest number of supervisions is returned.

```
1 MATCH (p:Professor)−[r:SUPERVISES]−>(:Person {Name: 'Stefan'})
2 RETURN (p), COUNT(r) AS NoOfSupervisions
3 ORDER BY NoOfSupervisions DESC
4 LIMIT 1
```
Listing 2.7: LIMIT Clause Example

CREATE

Say we are about to finish the master's thesis, but still need expert knowledge from a different academic domain to finish the thesis and thus need a third professor to give feedback on the content of the thesis. We want to create a new professor to give feedback as shown in figure 2.3.

To do so, we actually need to create the node and the relationship, so we have a two-step query. First, we create the new professor node:
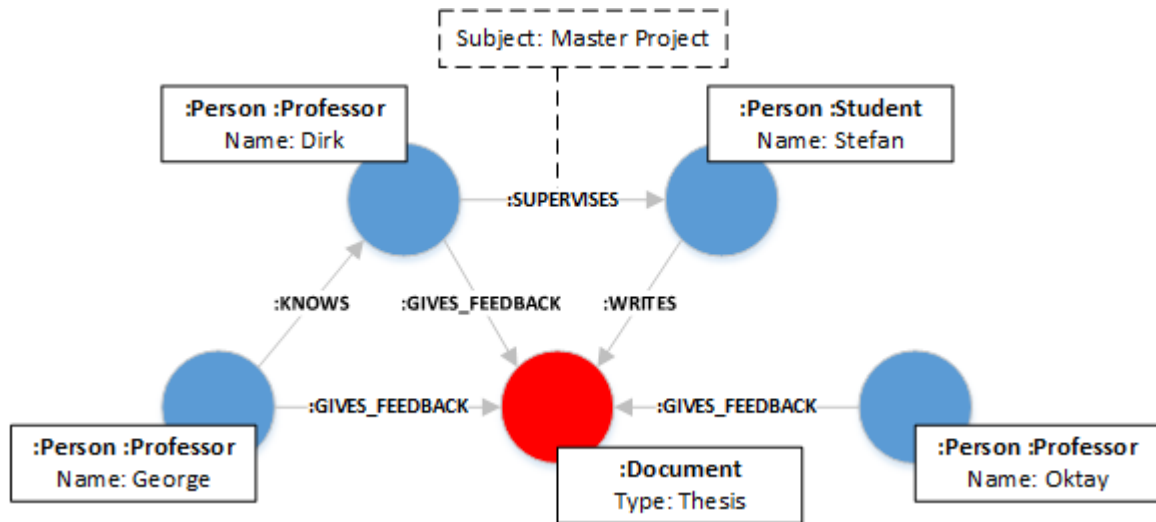
Figure 2.3: Graph Instance Example Extended

```
1 CREATE ( : Professor : Person {Name: 'Oktay'})
```
Listing 2.8: CREATE Node Example

Then we create a relationship *,:GIVES_FEEDBACK*, between the professor and the thesis:

```
1 MATCH ( p : Professor {Name: 'Oktay'})
2 MATCH( d : Document )
3 CREATE ( p ) −[:GIVES_FEEDBACK]−>(d)
```
Listing 2.9: CREATE Relationship Example

As you can see, we first need to match the two nodes we want to relate and can then create the desired relationship between the two. "MATCH" clauses can be defined after each other to include the sub sets of each query in the output.

MERGE

The "MERGE" clause is important in cases where certain nodes (or whole patterns) appear more than once in the returned set of objects. By using the "MERGE" clause in place of the "CREATE" clause, we ensure that any created pattern is distinct, i.e. if a pattern exists already, no new object is created.

Let us consult the example in listing 2.3 once more, where we used the "DISTINCT" function to avoid redundant Professor nodes in the output for professors with more than one student named Stefan under their supervision. This time, with "MERGE", we actually want to create a second relationship *:KNOWS* between the person and the professor. The query looks as follows.

```
1 MATCH ( p : Professor )−[:SUPERVISES]−>(s : Person {Name: 'Stefan'})
2 MERGE ( p )  −[rel :KNOWS]−> ( s )
3 ON CREATE SET rel . Status = "recently"
4 ON MATCH SET rel . Status = "a while"
```
Listing 2.10: MERGE Clause Example

For "MERGE" we can furthermore define different actions for when a pattern already exists, or not at the time of the "MERGE" execution. To explain this concept, we included the "ON CREATE SET" and "ON MATCH SET" clauses which are optional to the "MERGE" clause. "ON CREATE SET" lets us define the behaviour of the query when the pattern in the "MERGE" clause did not exist before, i.e. the query creates a new *(p) -[rel:KNOWS]-> (s)* relationship and set a property *Status* to "recently". "ON MATCH SET" covers the case that *(p) -[rel:KNOWS]-> (s)* already exists and only changes the "Status" property value of the existing *:KNOWS* relationship.

LOAD CSV

"LOAD CSV" is the clause that enables Neo4j to load data from CSV files. We made use of this command to bulk load the event logs into Neo4j in one go. The built-in function to load CSV-formatted files was the main determinant for the input format of our case studies.

Operators & Functions

Cypher comes with a number of operators and built-in functions. Next to mathematical operators such as "+" (addition), "-" (substraction), "*" (multiplication) and "/" (division), there are graph-specific operators as well. The *-operator, for example, allows us to specify graph patterns where we do not know how many relationships exist along the path.

```
1 MATCH path = ((p:Professor {Name: 'Oktay'}) −[∗]−(s:Student))
2 return path
```
Listing 2.11: Paths of Variable Length Example

In the listing above, we define a query that matches every student Oktay has a relationship to, no matter how remote this relationship is. The *-operator allows us to specify that we do not know the length of the path of a connection between two nodes. We implicitly introduced the capability of Neo4j to store graph paths in a variable.

Functions such as count() or sum() are only two examples of the functions implemented in Neo4j. Neo4j has many functions natively built in, such as predicate functions ("exists()"), or temporal functions ("duration()").

2.6 Database Schema

The term "database schema" has no clear standard definition and is ambiguously used. For example, a schema may refer to the data's actual, physical organization on a storage or to the logical associations and relationships of the data [34]. In Oracle database environments a database schema refers to the database objects created by a specific user, which essentially represents a specific view on of the entire database [1].

To disambiguate the term, we define database *schema* as follows: A database schema is the data structure of a database described in a formal language. The schema provides a blue-print to the data structure and allows to define integrity constraints as a set of rules to manifest intended behavior and restrict unintended behavior of the data. A visual representation of the essence of the described data model is part of the schema.

A database *instance*, the entirety of all database objects, implements a schema if it respects all of the schema's structural definitions and integrity constraints.

While for database concepts such as the relational database (RDB) can rely on a variety of standardized methods for data modeling and schema definition, such as the entity relationship model (ERM) [2], there exists, at the time of writing, no standard schema for LPGs.

Chapter 3

Schema Representation

In this chapter we take a close look into how a schema for a property graph can be defined. Our main difficulty with respect to schemata for our graph event logs is that no standard schema or data definition language (DDL) for property graphs exists. There are, however, approaches that cover our needs at least partly, such as the property graph based schema representation in Neo4j, i.e. the schema of the property graph is modeled as property graph as well. In [5] Bonifati et al. propose a schema language in which we found a good basis to build up upon to develop our own solution. W introduce a schema language in section 3.1 and a visual representation to complement the language in section 3.2. This schema definition can describe the graph data model attributes globally, such that it describes node types, relationship types, properties. It furthermore describes global attributes of properties such as the data type or uniqueness. We refer to the schema introduced in this chapter as *global schema*.

3.1 Schema Defintion

As property graphs are able to contain much information in just a few elements, a complete visual representation can be quite a challenge, especially if we want to abstract a schema from a graph instance. Actually, we want to be able to define a schema for graph instances in a way that we can add constraints and cardinalities to the different schema elements. As a first step towards this goal, we need to define a way to annotate a property graph schema. Therefore, we introduce a schema language, or DDL, for labeled property graphs based on the work of Bonifati et al.[5] which is based on OpenCypher and thus goes well together with our implementation with Neo4j and Cypher. They define a DDL that can describe the structural behavior of a LPG by listing all node types, relationship types and property types of a graph. To this proposal we add a unique attribute for property types.

For the global schema definition we assume

- $\mathcal{L}$ is a finite set of labels,

- $\mathcal{K}$ is a finite set of (property) keys,

- $\mathcal{T}$ is a finite set of data types,

- $\mathcal{BT}$ is a set of element types,

- $\mathcal{NT}$ is a set of node types,

- $\mathcal{ET}$ is a set of edge types.

A **property graph type** is a triple (BT, NT, ET).

A **property type** is a pair (k, t) with $k \in \mathcal{K}$ as property key and $t \in \mathcal{T}$ as its data type. For a property "Activity" of type STRING we write *Activity: STRING*.

An **element type** $b \in \mathcal{BT}$ is a 4-tuple (l, P, P_M, P_U) with label $l \in \mathcal{L}$, property types P, a subset of property types that are mandatory $P_M \subseteq P$ and a subset of property types with unique constraint on their values $P_U \subseteq P_M$, i.e. values of that property must be unique in a property graph instance. "Entity <u>ID</u>: STRING, EntityType!: STRING, Name: STRING" is an example declaration of element type $en = (\text{Entity}, \{pt_1, pt_2, pt_3\}, \{pt_2\}, \{pt3\})$ with $pt_1 = (\underline{\text{ID}}, \text{STRING})$, $pt_2 = (\text{EntityType!: STRING})$ and $pt_3 = (\text{Name: String})$. The <u>underlined</u> property keys denote the unique properties, the property keys with exclamation mark (!) denote mandatory properties and the property keys without additional features denote optional properties.

$prop(b) \coloneqq P$ defines the **property types** of an element type b. In other words prob(b) represents all property types of b. Similarly, $mand(b) \coloneqq P_M$ defines the **mandatory property types** of b, $uniq(b) \coloneqq P_U$ defines the **unique property types** of b and $label(b) \coloneqq l$ defines the **label** b possesses. Note that we limit the number of labels intentionally to 1. Even though property graph database implementations sometimes allow to assign multiple labels to a single node, we want to avoid this situation by definition. For a proper schema definition we require some reliable component to group the individual elements (nodes and relationships) to be able to apply constraints and cardinality definitions to those groups. If one element can be part of two groups, a clear schema definition becomes significantly harder.

A **node type** $nt \in \mathcal{NT}$ is a 1-tuple (b) with $b \in \mathcal{BT}$ as element type. *(:Event)* is the declaration of nt with b as element type and $label(b) = $"Event".

An **edge type** $et \in \mathcal{ET}$ is a triple (s, b, t) with element type $b \in \mathcal{BT}$, source node $s \in \mathcal{NT}$ and target node $t \in \mathcal{NT}$. Edge types, also referred to as relationship types in property graphs, are denoted as (:Event)-[:E_EN]->(:Entity) where $label(s) = $"Event", $label(b) = $":E_EN" and $label(b) = $"Entity".

With these definitions we can now define a complete, global property graph schema. As you might have notices, out graph definition only includes directed edges. This is because it is based on OpenCypher. We can, however, define an entity type for both directions and thereby allow bi-directional relationship in the property graph. On the other hand, Cypher allows to neglect the direction of a relationship type in queries, which effectively allows us to treat a relationship type as undirected, regardless of how it is declared in the global schema.

The following listing shows the global schema declaration with the schema language for the running example we introduce in section 4.1.

```
1  global_schema_running_example = (
2      {//element types
3          Event {
4              Activity!: STRING,
5              TS!: TIMESTAMP,
6              Amount: INTEGER,
7              Resource: STRING,
8              Offer: INTEGER,
9              # Terms: INTEGER,
10             Origin: STRING,
11             Case: INTEGER
12         }
13         Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
14         Log {ID}: STRING}
15         E_EN {},
16         L_E {},
17         DF {EntityTypes: LIST},
18         HOW {EntityTypes: LIST}
19     }
20     {//node types
```

```
21            (:Event), (:Entity), (:Log)
22        }
23        {//relationship types
24            (:Event)-[:E_EN]->(:Entity),
25            (:Log)-[:L_E]->(:Event),
26            (:Event)-[:DF]->(:Event),
27            (:Entity)-[:HOW]->(:Entity)
28        }
29 )
```

Listing 3.1: Running Example Global Schema Definition

We find the element types $\mathcal{BT}$ for nodes and relationships in the first section. The event got one property type per attribute in table 4.1, the log of our running example. Activity (Activity!) and the timestamp (TS!) property are marked as mandatory, because these two attributes are crucial to every event. The values of the property ID of *:Log* nodes must be unique in a graph instances which is based on this schema. Now that we have defined how we can declare a property graph schema with our schema language, we want to enrich our schema with a visual representation in the following section.

3.2 Visual Global Schema Representation

The schema definition serves as formal base to define a global property graph schema. However, in order to create a practical schema overview, we still need an easy to read representation that helps the user to quickly understand (or define) the structure of the graph. For this we also follow the suggestion of Bonifati et. al.[5] to use a property graph for the Visual representation of a graph schema. As we described in the preceding section, we utilize the concept of a labeled property graph itself to describe a labeled property graph schema. This can be done by raising the abstraction level of the graph itself, for example a node on the global schema level represents a node type while a node on instance level represents a node instance, or simply said a node. This node in turn has a label which is specified as node type in the graph schema. To be able to distinguish between graph instance and graph schema figure instantly, we change the appearance of the nodes such that they a node is represented as empty circle with a coloured line for schemata, as opposed to the a filled circle for instances. We specified in section 5.3.1 that every node of an instance, must have a label that is specified in the schema. The instance does otherwise not conform to the global schema. The same applies to relationships and properties.

Figure 3.2 shows the global schema for the running example. For the schema annotation we follow the same principle like for the graph instance annotation introduced in section 2.4. Nodes are represented as circles and the directed relationships are arrows pointing to the destination node. The labels are denoted, similar to the schema language, by a leading colon, e.g. *:Event* or *:E_EN*. The labels are written on their respective element types. Property types are placed in boxes with solid lines and the boxes are attached to the element types they belong to. For properties of relationship types, we chose boxes with dashed lines so that we can better distinguish between
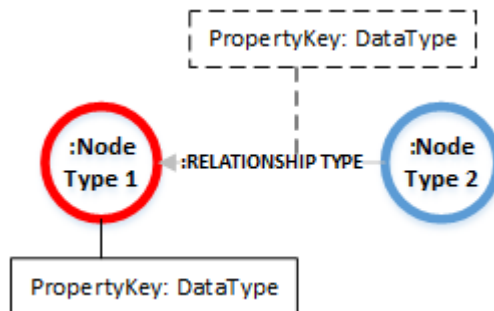
Figure 3.1: Visual Global Schema Elements
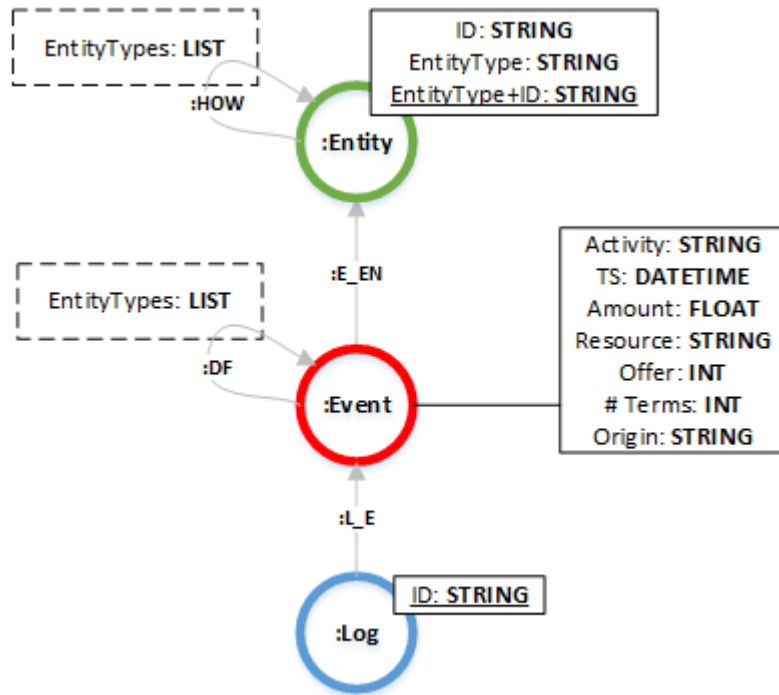
Figure 3.2: Running Example Global Schema

the two. If the graph schema grows and the number of properties confuses the graph overview by overlapping other schema elements, we allow to reduce the number of property types in the visual representation. In this case, a "..." shall be added to indicate that the representation is not complete and thus refer to the written global schema definition.

Chapter 4

Event Data in Labeled Property Graphs

In the last chapter we defined a schema representation for labeled property graphs. However, we still need a way to define a way to represent the event log elements and characteristics introduced in section 2.1 in a labeled property graph with such a schema. In other words, we need to find a way how we can, in general, transform events, activities, cases etc. into a labeled property graph instance such that this instance follows a given schema. In section 4.1 we introduce a running example. Section 4.2 discusses preceding research on the topic of event data in property graphs and in section 4.3 we introduce a set of transformation templates for event data in property graphs.

4.1 Running Example

To illustrate the concept of multi-dimensional event data, we introduce a simplified event log of a loan application process as running example. The example is based on the Business Process Intelligence Challenge 2017 (BPIC17) data set [32]. This data set also served as base for two of our case studies. For the running example we made some simplifications to the data shown, but the overall structure is still the same as in the original log. Table 4.1 shows a case of the running example.

We will consult this example case throughout the thesis in the context of multi-dimensionality of event data. It has a case identifier in the *cID* column uniquely identifying the loan application process instance. Furthermore, we have the different activities carried out during the process execution in the *Activity* column and a *Timestamp* column and some further case or event attributes. So far we have everything we expect from a minimal event log already, a case ID, activities and a

cID	eID	Activity	Timestmap	Amount	User	oID	Terms	Src
1	1	Create Appl.	29.08.19 10:30	1000	Raphael			A
1	2	Appl. Ready	29.08.19 10:35	1000	System			W
1	3	Create Offer	29.08.19 13:14	1000	Selma	1	128	O
1	4	Create Offer	29.08.19 13:49	1000	Patty	2	256	O
1	5	Send Offer	29.08.19 18:00	1000	Patty	2	256	O
1	6	Send Offer	29.08.19 18:00	1000	Selma	1	128	O
1	7	Offer Cancelled	30.08.19 13:49	1000	Selma	1	128	O
1	8	Offer Returned	30.08.19 13:49	1000	Patty	2	256	O
1	9	Appl. Complete	30.08.19 13:59	1000	Raphael			A

Table 4.1: Example Case Running Example

temporal order. *Amount* is an ordinary case attribute. The *User* column, representing the resource executing the task, can also be described as ordinary, even if it can be used to find organizational information of the process' operating organization, it is more or less a standard component of event logs. The resource attribute of an event usually describes the working resource that carried out the respective activity. This could be an employee, a machine or some virtual system component handling tasks of the process for example. The next three columns, however, can on one hand also be seen as ordinary event attributes, but on the other hand these columns contain much deeper information. Deeper in the sense that we basically find an ID value for offers in the *oID* column. The event attribute in the *Terms* column rather describes the offer than something specific to the loan application. An offer is actually a sub process of the application process and the *Src* column gives even more information about the structure of the application process. The values 'A', 'W' and 'O' indicate to what sub process an event belongs, i.e. the source of the event. In this case we have 'A' for application, the actual entity the event log has been created for, 'W' for the workflow information which usually originates from some workflow management system that supports the actual process and thus can be seen as an entity itself and the 'O' for offer which we discussed already. So we actually have three different types of entities involved in what we call a case. As discussed in section 2.2.1 we can actually split this case into as many sub-cases as we have entities involved and from these split entities we can construct the original case again, if we keep the information of the sequential order of events. Keeping the order information is necessary because most of the time the timestamp is not reliable enough in this respect, the two 'Send Offer' events for example could not be brought into their original order if we'd split the entities into separate logs and later merge them again. The example case in table 4.1 has one application entity and one workflow entity and these two entity types have a 1:1 relation in the log. The red markings in the table show that 'A' and 'B' can be identified over *cID* and the green marking shows the offer IDs. By combining the different ID types with their *Origin*, we can create unique identifiers for every single entity. Figure 4.1 shows the entity types and their cardinalities.
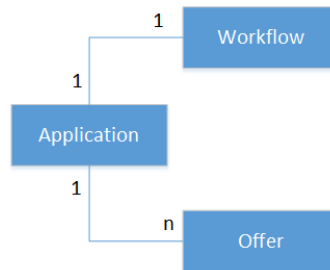
Figure 4.1: Running Example Entities

As described above, we have the three entities and between *Application* and *Offer* we have a 1:n relationship making our event data multi-dimensional. We will consult this simple example throughout the thesis to illustrate different concepts of graph databases and graph event data.

4.2 Previous Work

In [12] we introduced a preliminary representation of event data in a a LPG. This preliminary representation, however, had the sole purpose to help exploring the feasibility of storing and querying event data in LPGs and has not been developed with the purpose to get a general event log definition for label property graphs in mind. Figure 4.2 shows Neo4j's schema representation of that paper. The BPIC 17 data set has been used to create this schema and thus we can compare it very well to our running example. Please note that the figure is a standard output of Neo4j and is not in line with our visual representation annotations. The graph in figure 4.2 is a schema and would, in our annotation, contain nodes as thick coloured lines with non-coloured center.
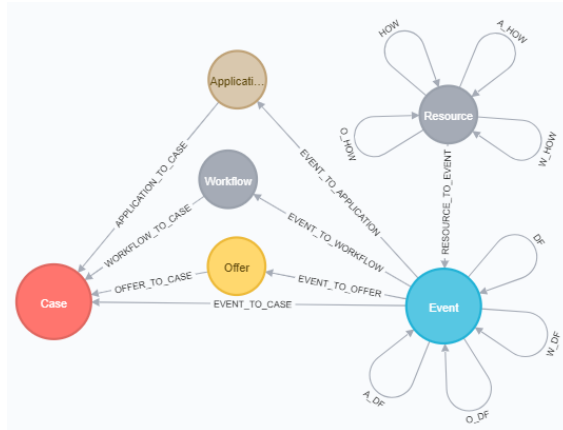
Figure 4.2: Previous Work Schema

A property graph generally offers many different ways to model the data schema. In figure 4.2, a rather intuitive and practical schema has been used to answer given process mining questions. The three entity types and the resources of the process became their own node types (their own label) and the case became the label case. For processes with increased complexity, however, this representation would lead to unreadable schemata. Therefore, we took this work as baseline to develop a more general and readable data model for graph event data.

The goal was to reach a concept to store event data in labeled property graphs, generic enough to allow the definition of compact, readable schemata for it and specific enough to enable the use of common process mining procedures and analysis on the data. In essence, we redefine the event log concepts described in 2.1 we developed for [12] and to enhance the entity concept described in section 2.2.1. We used entities as such in the paper already, see the *Application*, *Workflow*, *Offer* and *Resource* node types in figure 4.2, but not in a way such that we can generalize over all entity types. With entities, we also want to omit the definition of predefined cases for event logs, i.e. treat a case just as another business entity. The case in our running example log consists of the combined events of an application, a workflow and two offers. This means, if we find a way to only encode the application, workflow and offer events and make sure that we can relate these entities, we can define new cases from those low level processes. With low level process we mean a process that we cannot further divide into sub processes with their own identifiers. With such a data structure, we could for example define a new entity consisting of application and offer events only, or one that involves all three entity types which effectively reflects the case in the source log in table 4.1. This makes the concept of a case in events logs as we use it today obsolete and gives more flexibility to the analysts. Redefining a case with classical event logs always involved creating a new log, completely separated from the other case definitions of log.

4.3 Event Log Concepts

As outlined in the preceding paragraph, we want to develop a generic but still flexible conception of the event log concepts introduced in section 2.1. In the following paragraphs, we use the graph schema concepts from Chapter 3 to define graph notation for these event log concepts. Please note that we do not want the following graph concepts to be understood as immutable. A major goal of this research is to provide a basis that can be used to flexibly define event graph data structures according to the needs of given process mining questions. Therefore, the following definitions are just another set of possible graph representations of the existing event log concepts. In the following paragraphs we elaborate on how we translated the event log concepts to labeled property graph concepts.

4.3.1 Event

The core element of every event log is the event. Thus, it is the first graph event log element we define. Events are created by the execution of activities in some process that is supported by an information system. In a event log in CSV format, every event is represented as a row with the columns as attributes to that event. Generally, we can choose out of three graph element types: node, relationship and property. For the events, we decided to create a dedicated node type. Nodes are the actual data points in a graph database and they can be enriched by properties. For events, this is exactly what we need. In event nodes we are able to encode all event attributes in form of properties. Figure 4.3 shows such a node. The label for this node type is *:Event*.

Figure 4.3: Event Node

These event nodes, i.e. nodes with the label *:Event*, are the core concept for our approach, since they are the starting point for every event log and all further concepts can be based on events. In order to store an entire CSV log, we can add a property for each column to the event nodes, such that we can build the graph structures needed from the information encoded in the event nodes.

4.3.2 Activity

The concept of activity is basically some attribute that is specific to an event and the logical consequence is to model it as an attribute to events. Figure 4.4 shows an event from our running example with the visual property notation we introduced in 2.4.
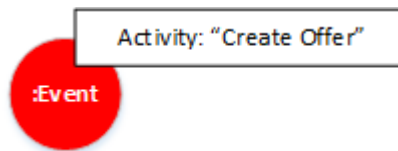
Figure 4.4: Activity

4.3.3 Timestamp

For the timestamp, the same reasoning applies like for the activity concept. Every event node has a timestamp property, like shown in the example in figure 4.5.

Figure 4.5: Timestamp

4.3.4   Event Log

The event log, in terms of the traditional process mining related definition, is a collection of events that are somehow related to a case identifier. If a data set has more than one case identifier, we usually have to deal with multiple event logs. Thus, some data sets include one event log while others include more. To cope with that, we introduce a separate node type *:Log* for the logs as shown in figure 4.6.

Figure 4.6: Log Node

So every event log of data sets we want to analyze together becomes its own *:Log* nodes to which the respective events can be correlated.

4.3.5   Case

The case is a primary concept in traditional event logs, since it predetermines from what perspective the event log is looked at. In sequential event logs, if the case definition changes, a new log must be created. In our graph-centric definition of an event log, the case identifier only plays a secondary role, because the focus is on entities of a process. This means our focus within a log is on events and their respective entities. Thus we use a generic *:Entity* nodes as shown in figure 4.7.

Figure 4.7: Entity Node

With these nodes we can represent all the different entities of a process, such as *Application*, *Workflow*, *Offer* and even the case identifier and the resource. This is one of main differences to the design in section 4.2. In the first approach we modeled every single entity as its own node type, which worked very well in terms of storing and querying the data. For a generic schema, however, this model would lead to unnecessary many elements and confusing schemata.

4.3.6   Attributes

In traditional process event logs we differentiate between case and event attributes, i.e. data fields that can logically be assigned to case or event level as introduced in section 2.1. In property graphs we can, similar to the timestamp and activity, make all attributes node properties. In the complete lack of domain knowledge, all attributes might become event properties. Figures 4.8 and 4.9 show how attributes can be added as properties to the different elements. The event attribute in the graph shown in figure 4.8 is not much different from the tradition event attribute.

The case attribute, however, becomes more diverse because of our notion of entity which we introduced in section 2.2.1. Thus we call case attributes in our property graph design entity attributes and these attributes can vary between the different entity types. In figure 4.9 we added the "# Terms" attribute we find in our example table 4.1 to an *:Entity* of type "Offer".

As you can see, because of the generic *:Entity* node, we need some additional means to differentiate between the entity types we have in the log. In our case, we added another property called "EntityType" to the *:Entity* node. This, however, is a design decision left open to the creator of
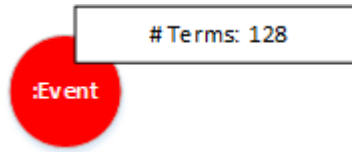
Figure 4.8: Event Attribute
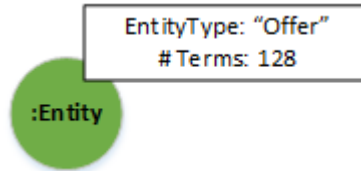
Figure 4.9: Entity Attribute

a graph event log, since the property graph data model offers many different ways to realize such a differentiation.

Similarly, we can add attributes to logs.

4.3.7 Sequential Order

The sequential order is more an implicit concept of event logs, as the order is usually defined by the timestamps. While the question whether event a and event b happened one after another is not our concern at this point, we still want to define how we annotate a directly follows relationship between events a and b. As shown in figure 4.10, we use a directed relationship between two event nodes for that.

Figure 4.10: Sequential Event Order

4.3.8 Further Event Log Concepts

We introduced the most basic concepts of event logs in this chapter. Most more advanced concepts of event logs or process mining build upon these concepts, for example a handover of work social network of resources in the field of organizational process mining. At this point we do not intend to define graph annotations for any of the advanced event log concepts, since our goal is to develop a framework that enables us to formally define these concepts for every event log individually and include them in a schema representation accordingly.

Chapter 5

Schema Framework for Graph Event Data

In the last chapter we demonstrated a way to represent process event data in property graphs and in Chapter 3 we showed a recent schema approach for property graphs. However, with this global schema definition we cannot define constraints or rules for encoding behavioral properties in graph data. In this chapter we propose a modular schema modeling approach for property graph schemata. We introduce this framework as a set of concepts that constitute a way to define a property graph schema with rules and constraints to restrict property graph instances using the schema. Section 5.1 explains the problem an our idea on an example. In section 5.3 we introduce a schema language for local pattern schemata and a complementary visual representation. In section 5.4 we introduce consistency rules to govern the data structures of graph instances which are based on the schema patterns. Section 5.5 wraps up how the different concepts in our framework combine to create a (partial) schema for a property graph, we refer to as local schema.

5.1   Defining More Restrictive Schemata

In the set of figures below we show a simple global schema (figure 5.1) and show two example instances, one with a data structure we expect for graph event data (figure 5.2) and one instance that does not go along with our expectations (figure 5.3), even though both graph instances conform to the global schema in figure 5.1.

The global schema in figure 5.1 specifies two node types, *(:Event)* and *(:Entity)*, and two relationship types, *(:Event)-[:DF]->(:Event)* and *(:Event)-[:E_EN]->(:Entity)*. The instances in the figures 5.2 and 5.3 also only include objects of these four object types and thus conform to the global schema. In the graph in figure 5.2 all event have their timestamps as required and the *:DF* relationships are well in order with the *Timestamp* properties of the *:Event* nodes. Every event is associated with the entity *en1*. In addition, the following properties of graph event data hold: The clear association of events with directly follows relationships to entities; and keeping the temporal order of evens in a relationship $(e1{:}Event)\text{-}[{:}DF]\text{-}{>}(e2{:}Event)$ such that $e1$.Timestamp $\leq e2$.Timestamp. The behavioral properties encoded in the graph in figure 5.3 are inconsistent with the attributes, making reliable querying on event data impossible. For the *:DF* relationship to be defined well, it cannot just be specified in the global schema definition of Chapter 3, we have to add further constraints, such as a temporal order dependency of *:DF* relationships, over an instance. For example, *:DF* relationships should not start and end with the same node (no self loops), the temporal order of events should be reflected by the directions of the *:DF* relationships (*e1* does not follow *e2*) and there should also be at least one relationship between an entity and an event (*e3* to *en1*). Furthermore, the *:DF* relationship between *e1* and *e2* should not exist, because there is no single entity node related to both event nodes. These examples make the need for more specific structural constraints very clear.
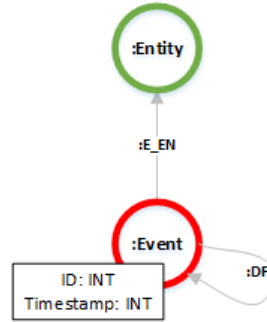
Figure 5.1: Simple Example Global Schema
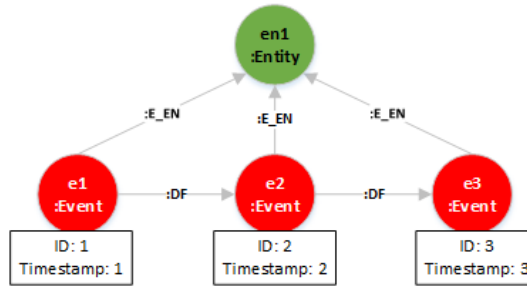
Figure 5.2: Example Instance with Desired Structure
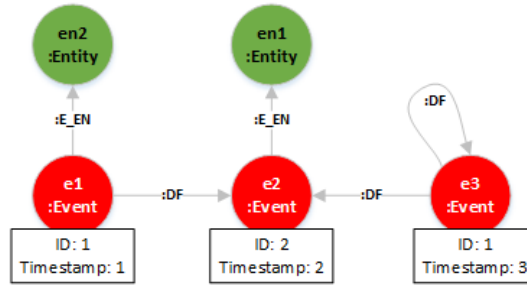
Figure 5.3: Example Instance with Undesired Structure

The second problem with respect to defining structural constraints on property graphs is the definition of such rules itself. As we have shown in the small example above, property graphs can be very diverse and complex even with a small number of objects. Thus, the definition of constraints for an entire graph schema is not feasible as a very complex set of rules would be required.

Therefore, we need a way to take the global schema definition from Chapter 3 and the consistency rules to model it in accordance with our domain-specific requirements. As the constraint definition to such a schema is too complex, we want to reduce complexity by defining rules on local patterns only and by splitting the schema into sub-schemata, or patterns of the schema, to reduce the complexity and enable the definition of constraints and other rules.

5.2 Requirements and Proposed Solution

To identify the requirements for the rules, we analyzed and iteratively developed schemata for 5 event data sets described in detail in Chapter 7 and appendix A. We eventually identified that we need to restrict instances of a schema in the following ways:

1. Make property keys mandatory for specified node or relationship types. *:DF* relationships,

for example, must indicate for which entities they apply.

2. Limit property values to be globally unique in an entire graph instance. For instance, ID properties of entity nodes.

3. Limit the number of possible materializations of specific object types. For example there shall be only a single relationship to correlate an event *e1* with an entity *en1*.

4. Define cardinalities between node types, e.g. events can only relate to one log.

5. Make the materialization of nodes and relationships dependent on the existence or non-existence of graph object types. For example we can only create a *:HOW* relationship if a corresponding *:DF* relationship exists.

6. Make the materialization of nodes and relationships dependent of properties and labels of other objects. The direction of a *:DF* relationship, for example, depends on the *Timestamp* properties of its source and target nodes.

From these above requirements, we developed the following proposal for specifying schemata for event data in graph databases: There are two types of a schemata. 1) A *graph schema* is a 1-tuple (Global-Schema) that defines all possible node types, relationship types and element types with property keys and data types. 2) A *local schema* is a pair (Local-Schema-Pattern, Rules) that defines for a subset of the node types and relationship types of the global schema to be mandatory in the global schema and add additional constraints. Local Schemata can build on the global or other local schema by inheriting consistency rules already defined and adding additional consistency rules as shown for the local pattern in figure 6.1 for example. The local pattern in figure 6.1 makes *:Log*, *:Event* and *:Entity* nodes; *:L_E* and *:E_EN* relationships and their property types mandatory. The rules to that pattern defined in section 6.1 for example specify that every event node must be the source node of at least one *:E_EN* relationship, because the event would not have any 'case notion' we could assign to it. This rule falls under requirement 1, as it is concerned with cardinalities between event and entity nodes. We can extend the core pattern by adding the directly follows pattern in figure 6.4. As stated above, the new local pattern then combines all element types and rules of the two patterns.

The requirements 1-7 fall into 2 categories: 1) global structural requirements we can add to the existing schema language (by extending), 2) local structural requirements which restrict for example when two or more nodes may be in a relation based on the properties of these nodes.

The first category, where requirements 1 and 2 fall in, we can cover entirely with the schema language in section 3.1. Requirement 1 can be satisfied by the definition of a property type to be mandatory denoted as "PropertyKey!" in the schema language. Requirement 2 can be satisfied by specifying a unique property type as "PropertyType" accordingly.

For the second category we propose the following idea: we specify the structure for which relations between nodes must be restricted further as local schema pattern in terms of LPGs and then specify consistency rules that must hold over this local schema in terms of logical constraints, any set of nodes and relations that matches schema patterns must satisfy the corresponding consistency rules, just as in the example schema in figure 5.1, for which we specified that for (e1:Event)-[:DF]->(e2:Event), $e1.\text{Timestamp} \leq e2.\text{Timestamp}$ must hold.

So in total that means that each schema is defined in terms of LPGs, defining node types, relations, properties. The local schema has in addition consistency rules that have to hold for all element types matching the schema, i.e. pairs of (Local-Schema-Pattern, Rules) define sub-schemata of a graph schema (Global-Schema).

5.3 Pattern Schema Definition

In this section we explain how local schema patterns can be defined. A pattern definition consists of two parts, a formal definition in the schema pattern language we introduce in section 5.3.1 and a visual representation introduced in section 5.3.2.

5.3.1 Local Schema Pattern Language

As we showed in the previous paragraph, the global schema representation introduced in Chapter 3 is suitable to define element types a graph instance can contain, but it cannot be used to restrict or prescribe how specific local patterns shall be shaped in an instance as specified in requirements 1-6. Chapter 3 defines a *schema* as a triple (BT, NT, ET) which can be understood as a global pattern for the entire graph. For the definition of local patterns we assume the following to be given:

- $\mathcal{L}$ is a finite set of labels,

- $\mathcal{K}$ is a finite set of keys,

- $\mathcal{BT}_P$ is a set of pattern element types,

- $\mathcal{NT}$ is a set of pattern node types,

- $\mathcal{ET}$ is a set of pattern edge types,

- $\mathcal{GP}$ is a set of property graph patterns,

We define a schema as *set* of pairs (gp, rd) where each *local property graph pattern gp* is 4-tuple $gp = (BT_P, NT, ET, GP)$ that can inherit from other patterns, and rules (defined in section 5.4) further restrict instances w.r.t. the node types NT and edge types ET in gp.

A **pattern property type** is a 1-tuple (k) with $k \in \mathcal{K}$ as property key. We do not specify data types in patterns as they are specified in the global schema already, we thus do not need a second component t to the pattern property type. We, however, use the same set of keys $\mathcal{K}$ like for the global schema definition in section 3.1.

A **pattern element type** $b \in \mathcal{BT}_P$ is a triple (l, P, U) with label $l \in \mathcal{L}_P$, P is a subset of pattern property types with unique constraint $P_U \subseteq P$. For example, the pattern element type for an event with activity and timestamp and a unique ID may be defined as **Event {ID, Activity, Timestamp}**. The unique property types are underlined. Mandatory properties are not specifically denoted since all property types in patterns are always mandatory.

A **pattern node types** $(nt \in \mathcal{NT})$ and a **pattern edge type** $(et \in \mathcal{ET})$ are defined exactly the same way as the schema node types and schema edge types in section 3.1.

A **property graph pattern** $gp \in \mathcal{GP}$ that inherits a pattern gp', receives all definitions of the inherited pattern. This implies $BT' \subseteq BT, NT' \subseteq NT$ and $ET' \subseteq ET$, i.e. gp inherits all pattern element types, pattern node types and pattern edge types of gp'.

The local patterns have a higher abstraction level compared to the schema in Chapter 3 and thus we need a modified language and representation. We again pick up the work of Bonifati et. al. in [5] and adjust it to define a schema language for the patterns accordingly, a pattern schema language. The main adjustment is the introduction of inheritance of patterns (sub-graphs). We also drop the $mand()$ function, since for patterns we define that every element type of a pattern as mandatory for the schema, i.e. a pattern is always a subset of a schema. We want to keep the event log creation as flexible as possible, hence patterns do not prescribe data types, but if there are specific requirements to property values they can be specified in the rules. A pattern property can be defined with a unique constraint, which effectively tags the respective schema property as unique as described in section 3.1. The pattern property unique constraint is rendered as underlined property key in the language as well as in the visual representation. The schema pattern language is in essence is a generalized version of the schema language.

5.3.2 Visual Pattern Representation

Now that we have defined a way to represent a pattern in a written language to fully define every aspect of a pattern, we want to add a visual representation to accompany the written definition. This is to represent the patterns in an intuitive, quickly readable format, i.e. we again use a

property graph as we did for the schema, but with slightly different semantics. The full visual representation and its semantics is discussed in the following paragraphs.

Pattern node types and pattern relationship types come in the same form as the schema node types and the schema relationship types: as a circle and a directed arrow and a label indicated by a leading colon. This label is equal to the pattern element's type. Their respective pattern property types are written in boxes attached to the element. Property types of node types are denoted in boxes with solid lines and the pattern relationship property types in boxes with dashed lines. We use the full capitalization of the relationship types (:RELATIONSHIP) just as in the schema representation to help the reader clearly distinguish node (:Node) and relationship labels. This notation is commonly used with the graph query language Cypher [14]. Figure 5.4 gives an overview with all visual components in an example pattern.
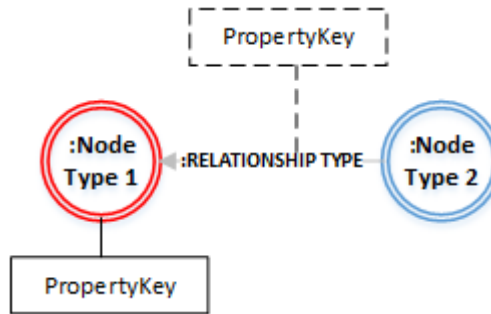
Figure 5.4: Pattern Element Types Overview

The difference between the visual pattern and the visual schema representation (figure 3.1) is that the patterns do not specify data types for property types. As we also defined in the schema representation in section 3.2, different pattern node types can be coloured differently in the representation. This colouring comes with no semantics, but is intended to help the reader to better and faster understand the differences within a pattern, but even more to better understand the relations between the different layers, i.e. instance, schema and pattern. The latter requires consistent colouring of the related elements of course. The relations of elements of the different layers is closer discussed in the following section.

5.4 Consistency Rules

As specified in the requirements in section 5.2, we need a way to govern the graph data structure on instance level. During the analysis of five different data sets we identified and used the following types of consistency rules:

- Limit cardinality of an outgoing relationship of a node type n: $|(n) - [] - > ()| \underset{>}{\overset{\leq}{\lessgtr}} k$ must hold

- Limit cardinality of an incoming relationship of a node type n: $|() - [] - > (n)| \underset{>}{\overset{\leq}{\lessgtr}} k$ must hold

- Limit relationships (rel) based on property values of nodes: for any $(n) - [rel] - > (n2), n1.x \underset{>}{\overset{\leq}{\lessgtr}} n2.y$ must hold

- Limit relationships based on the label of nodes: for any $(n) - [rel] - > (n2), label(n1) = / \neq x$ must hold

- Limit relationships (rel) or nodes (n) based on property values of remote relationships: for any $(n) - [rel] - > () - [rel2] -, rel2.x \underset{>}{\overset{\leq}{\lessgtr}} k$ must hold

- Limit relationships (rel) or nodes (n) based on the label of remote relationships ($rel2$): for any $(n) - [rel] -> () - [rel2]-, label(rel2) = / \neq x$ must hold

- Limit relationships (rel) or nodes (n) based on the existence of remote relationships: for any $(n) - [rel] -> () - [rel2]-, rel2$ must exist

- Limit relationships (rel) or nodes (n) based on the existence of remote relationships: for any $(n) - [rel] -> () - [] - (n2), n2$ must exist

The list may be extended by further consistency rules as event data with further requirements is to be encoded and analyzed in a property graph. The types of consistency rules specified above, however, are sufficient to encode the behavioral attributes to graph event data of all five event logs.

For the rules, we use Cypher syntax combined with regular predicate logic and precise language, e.g. $|(:Log)-[:L\_E]->(e \in \mathcal{N}:Event)| \leq 1$ defines that any *Event* node of the instance may have a maximum of 1 *:L_E* relationships. Since the patterns strictly define the graph schema elements, it is not necessary to declare node types of source and destination nodes if the relationship type is included. If one of the patterns used has a relationship type $(Log) - [L_E] -> (Event)$, we can shorten the rule definition for the instance (based on the pattern) to $|() - [: L\_E] -> (e \in \mathcal{N})| \leq 1$ since the information that the source node must be of type *:Log* and the destination log must be of type *:Event* is given by the pattern definition.

5.5 Framework Overview

The patterns and their rules are the core components to the schema framework approach we propose in this thesis. They enable us to flexibly adjust the schema to the manifold and changing requirements on process event data while preserving the data structure we require for specific applications or algorithms. For example, as we showed above where we need to ensure that our graph has the structure in figure 5.2 and as opposed to the data model shown in figure 5.3. A local pattern with *:Event* node type and *:DF* relationship type could for example be defined on the global schema in figure 5.1, with a rule to ensure that $e1.Timestamp \leq e2.Timestamp \leq e3.Timestamp$ holds. This local schema alone is sufficient to invalidate the graph in figure 5.3.

Note that a global schema may include many more components than defined in its local patterns, but what's included in the patterns and their rules is the essential minimum without which the vehicle was not functional.

A schema contains every aspect a graph instance can possess, i.e. node types, relationship types and property types. The patterns prescribe the minimal subset of graph elements of the schema to make sure it can describe the right type of data and the rules define constraints and requirements to the graph instance to ensure the correct structure. Just as a graph instance must conform to its schema, as stated in the last paragraph, a schema must conform to its patterns and the instance must conform to the rules. A schema conforms to its patterns if it contains at least all graph elements defined in the patterns. An instance conforms to the rules if not a single rule, e.g. a cardinality condition or integrity constraint, is violated. Figure 5.5 gives an overview of the framework and how the components relate to each other. In essence, a pattern is a partial schema for a property graph schema.

In the framework, want to be able to verify three things. First, if a particular graph instance conforms to a given global graph schema, i.e. only element types defined in the global schema may be used in the instance. If an instance contains any different element it does not conform to the schema anymore. Second, if a particular global schema conforms to the given local schema, i.e. all elements of the pattern of a local schema are also included in the global schema. Third, if a database instance conforms to the pattern rules, i.e. no pattern rule is violated by the data structure of the graph instance.

Every pattern comes with its own set of rules. The link between rules and patterns is necessary, because different requirements on the graph event data, e.g. for certain process mining techniques,
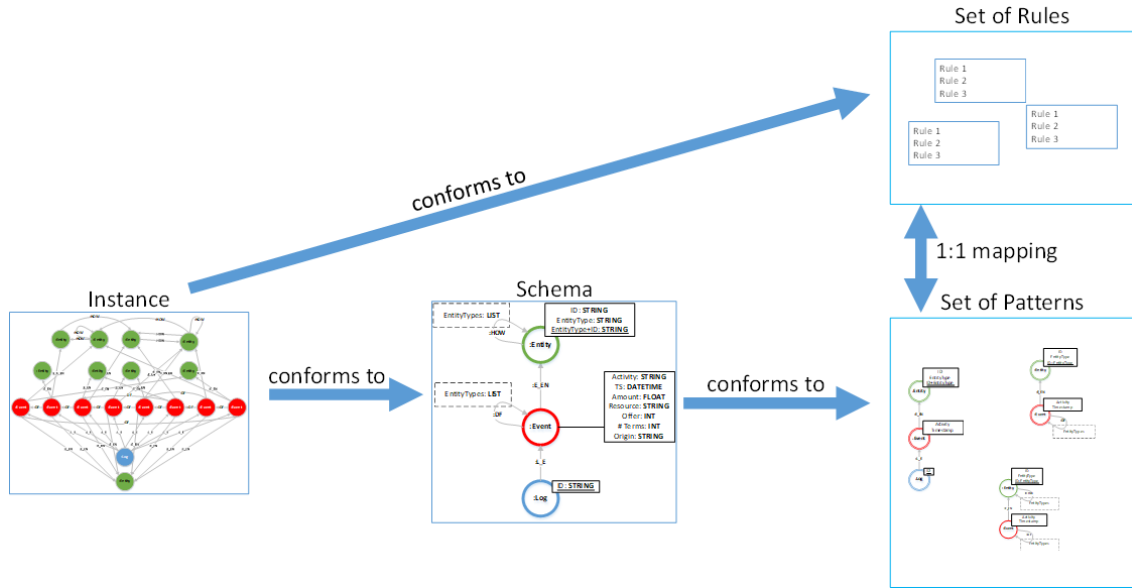
Figure 5.5: Framework Overview

come with different expectations to the graph data structure. So even if the two patterns are of equal structure, they can differ in terms of their rules. To be able to design dependencies between patterns, we introduced pattern inheritance to the definition of a pattern. The pattern definition is in essence the schema of a pattern as described in section 5.3.1. This way we are able to create a hierarchy of patterns that build up on each other, i.e. the inherited pattern is required for the inheriting pattern.

Figure 5.6 shows the hierarchical relationships of the patterns created for the case studies, i.e. a pattern that defines the core structure (0_core) of all event logs, a directly follows pattern (1_df) to create the sequential order of events, a pattern for the handover of work (2_how) between resources, a pattern for coinciding entities (1_en_coincide) in a single or over multiple logs and a similar pattern for coinciding events (1_e_coincide). The numeration in the pattern names suggest their level in the hierarchy already. Each pattern always has at least one parent in each upper layer, but since patterns on one layer are independent of each other not all patterns of a layer must be inherited by a pattern of a lower layer. However, this framework approach leaves the graph event data design open, i.e. it allows to freely define new patterns if needed such that a different set of event data may have even a different core pattern and thus would not use a single pattern introduced in this work.
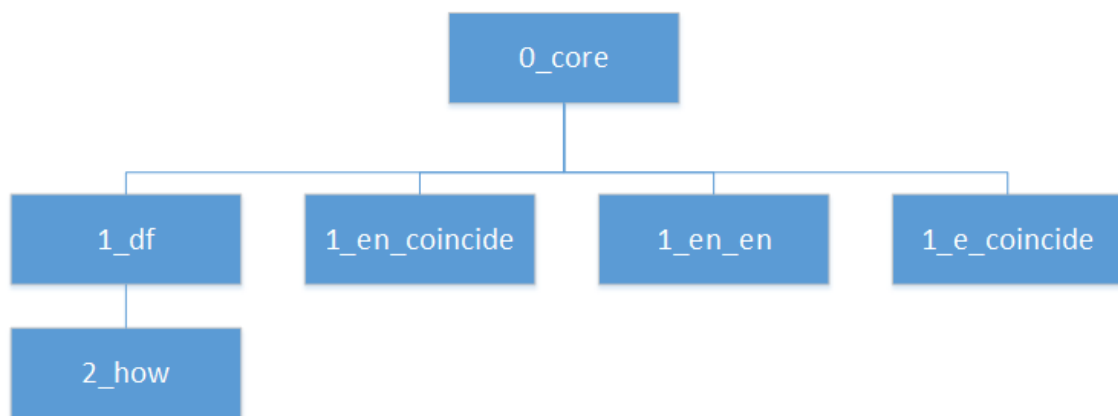
Figure 5.6: Pattern Hierarchy Overview

Chapter 6

Schema for Graph Event Logs

In this chapter we combine the local pattern definition introduced in the previous chapter, combined with the global schema definition from Chapter 3 to develop example schema patterns templates for the five data sets w.r.t. the event log concepts introduced in Chapter 4 to create the local patterns shown in figure 5.6. In section 6.1 we introduce the pattern that basically defines the core structure of graph event data, thus it is called *0_core*. Section 6.2 introduces the local pattern for directly follows relationships between events. In section 6.3 we define a pattern that copes with multiple events that actually originated from the same activity. This pattern can, for example, be used to tackle the data divergence problem. A similar pattern, but for coinciding entities, is introduced in section 6.4. Section 6.5 introduces a pattern for the handover of work concept, used for organization-related process analyses. In section 6.6 we discuss how such a schema, local patterns and graph instance can be verified as illustrated in figure 5.5. Section 6.7 provides a summary to conclude the chapter.

The patterns proposed in this chapter are one example of many ways to define graph data structures for event data. Different data sets or analysis questions may require an entirely different set of patterns.

6.1 Event Log Graph Core

This pattern template is the starting point for any graph event log that is created according to this work. The core pattern template includes all event logs concepts as required in process mining, e.g. by Van der Aalst [27].

The central element, as the name event log suggests, is the event and this is not different in graph event logs. An event must at least have an *Activity*, which is a general description of what has been done, and some attribute for temporal ordering of events, usually found as timestamp or the sequence in which events are recorded in the source information system. This information is necessary to recreate the order of events from the original log in the graph. The *Timestamp* property of *Event* nodes account for this. Note that this template does not include the sequence information of events, since this can be interpreted in different ways (e.g. strict sequence or partial order) we define a separate template in section 6.2. Different definitions of directly follows can be introduced as additional patterns. Next to *Event* nodes, the core pattern template contains a *Log* node type to enable us to store multiple event logs in a single graph instance. A *Log* node must have a unique *ID*. The *Entity* node type is based on the concept of an entity which has been introduced in section 2.2. An *Entity* node must have an *ID*, an *EntityType* and a unique key which is essentially a combination of *ID* and *EntityType*. This composite key construction is necessary, at least for our application, because we use entities to replace the strict notion of a case. The concept of a case is commonly used in traditional event logs as described in section 2.1. If the data permits, multiple entities can be derived from one specific case which would result in entity nodes having the same identifiers. To illustrate this we consult our running example once more.

The Origin column in table 4.1 show the letters "*A*", "*W*" and "*O*" standing for "*Application*", "*W*orklfow" and "*O*ffer" respectively. We find a separate id column (*Offer*) for the offer entities as we can see in the green marking in table 4.1. For application and workflow entities, however, there is only the case identifier and thus we cannot differentiate between the two entities by an existing ID, as we can see in the red markings. Thus we can use the combination of the case identifier and the origin, or entity type, to make the entities uniquely identifiable. This construct is, of course, a necessity that comes from the structure of the data source we used and this would probably be different for different data sets or sources. Logs are connected to events by *L_E* (log to event) relationships and events are connected to entities by *E_EN* (event to entity) relationships, both with no properties.

The following paragraphs define the pattern and its rules in more detail.

Pattern: Figure 6.1 illustrates the visual representation of the pattern.
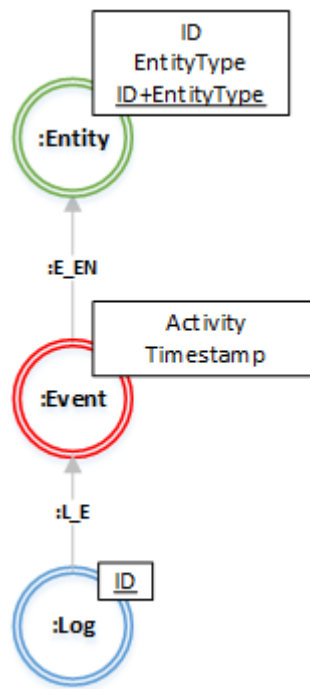
Figure 6.1: Graph Event Log Core Pattern

The core pattern brings the event log characteristics we defined in Chapter 4 together to form the most basic representation of graph event data. Depending on the complexity of the pattern, the visual representation must be simplified to remain readable, e.g. by removing properties from the node types. For this reason we introduced the schema pattern language in section 5.3.1. The definition of the core pattern in the schema pattern language looks as follows:

```
1  0_core = (
2      {//element types
3          Event {Activity, Timestamp}
4          Entity {ID, EntityType, ID+EntityType},
5          Log {ID},
6          E_EN {},
7          L_E {}
8      }
9      {//node types
10         (:Event), (:Entity), (:Log)
11     }
12     {//relationship types
13         (:Event)−[:E_EN]−>(:Entity),
```

```
14              (:Log)−[:L_E]−>(:Event)
15         }
16      {//inherited patterns
17
18         }
19  )
```

Listing 6.1: CORE Pattern Definition

We have three node types and two relationship types in this pattern. Each of which classifies a distinct subset of nodes and relationships respectively. As this is the core pattern which forms the base for all logs, we have no inherited patterns. It is the base for all further patterns we developed for this thesis. Since the patterns and rules only work together, we need to create a link between them. The following sets of nodes and relationships of the graph instance can be affected by the rules of this pattern, because they are part of the pattern definition:

- *Event* nodes $\mathcal{N}_e = \{n|n \in \mathcal{N} \wedge "Event" = label(n)\}$, which map to the *(:Event)* pattern node type.

- *Entity* nodes $\mathcal{N}_{en} = \{n|n \in \mathcal{N} \wedge "Entity" = label(n)\}$, which map to the *(:Entity)* pattern node type.

- *Log* nodes $\mathcal{N}_l = \{n|n \in \mathcal{N} \wedge "Log" = label(n)\}$, mapping to the *(:Log)* pattern node type.

- The 'log to event' relationships, *L_E*, $\mathcal{R}_{le} = \{r|r \in \mathcal{R} \wedge "L\_E" = label(r)\}$, mapping to the *(:Event)-[:E_EN]->:Entity)* pattern relationship type.

- The 'event to entity' relationships, *E_EN*, $\mathcal{R}_{een} = \{r|r \in \mathcal{R} \wedge "E\_EN" = label(r)\}$, which map to the *(:Log)-[:L_E]->(:Event)* pattern relationship type.

Now we have defined the sets of nodes and relationships on instance level that are affected by rules. Schema node types, relationship types and property types of the schema level have equal names like the pattern's node, relationship and property types. Figure 5.5 helps to grasp the different layers and how they relate to each other.

Rules:

1. $\mathcal{N}_e$, $\mathcal{N}_{en}$ and $\mathcal{N}_l$ are disjoint with any set of nodes.

2. Every event node e has maximum one L_E relationship: $|() − [: L\_E]− > (e)| \leq 1$

3. Every log node l has at least one L_E relationship: $|(l) − [: L\_E]− > ()| \geq 1$

4. Every event node e has at least one E_EN relationship: $|(e) − [: E\_EN]− > ()| \geq 1$

5. Every entity node en has at least one E_EN relationship: $|() − [: E\_EN]− > (en)| \geq 1$

The base pattern rules only describe the fundamental prerequisites of a graph event log. Rule 1 states that the set of nodes of the node types of this pattern are disjoint with any other set of nodes of this instance, including all node types that are included in the schema, but not in the schema patterns. Rules 2 to 5 are concerned with cardinalities. Rule 2 defines that an event node can have maximum one log. Usually we expect event nodes to have a log, but there can be exceptions, for example event nodes that function as collector node for coinciding event nodes across logs or within a log. Rule 3 defines that a log node has at least one event. This is rather intuitive since an event log without events is not very useful. Rules 4 and 5 are concerned with the cardinality of relationships between events and entities. An event always needs a case (entity) and every entity always needs at least one event, otherwise neither of them would fulfill the requirements of event data as defined in section 2.1. So rules 4 and 5 define the lower bound of the n:n relation between event and entity which gives us [1..n]:[1..n]. At this pattern, every event

node needs to be (directly) related to at least one entity and vice versa. We later define patterns that manipulate the last two rules by introducing special versions of event node types and entity node types such that the lower bound of the relation between the two node types may change to [0..n]:[0..n], but only for those special cases of so called collector nodes for coinciding events and entities as mentioned in the pattern hierarchy in figure 5.6 already. We elaborate more on that in the respective pattern descriptions below. Even though there is not much information to be mined from an instance based on the core pattern only, the basic structure for the graph event data is already set and we could implement a graph event log without any further pattern.

Examples: To illustrate the concepts of the patterns, we show some simple examples on how a pattern can materialize in a graph instance. All examples show sub-graphs of a graph event log on instance level, i.e. we are talking about actual nodes rather than about node types. We use the graph instance annotation as introduced in 2.4 together with a Cypher-like declaration of variables, e.g. (e:Event) written in a node means *e* represents the node with the node type *:Event*. With *e* we can address the properties of a node just like in Cypher, e.g. *e.Activity* refers to the event nodes *Activity* property.

Figure 6.2 shows three entities *en1*, *en2* and *en3*, two events *e1* and *e2* and a log *l*.
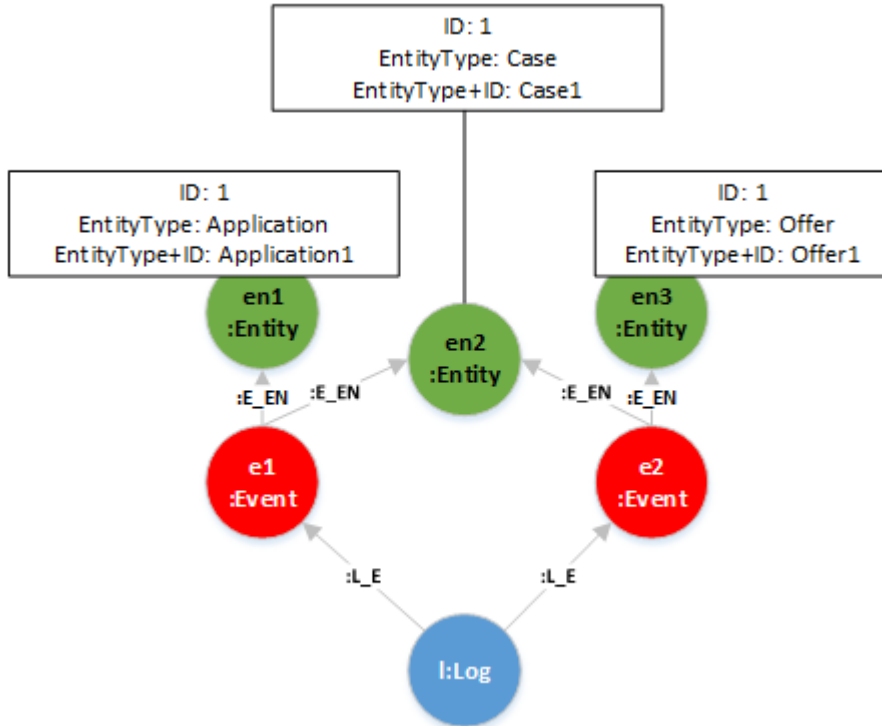
Figure 6.2: Core Pattern Conform Instance

The core pattern primarily defines cardinalities of the different graph elements. The example above respects rule 2, because the two events have only one *:L_E* relationship each, which in turn also satisfies rule 3 since we do not have a log without *:L_E* relationships. Rules 3 and 4 are satisfied as well because all *:Entity* nodes and all *:Event* nodes have at least one *:E_EN* relationship and all relationship directions are correctly defined as required by the rules.

The following example in figure 6.3 does not conform to the rules. The new example has now two log nodes *l1* and *l2*. The issue of this sub-graph w.r.t. the rules is that event *e2* is related to two different logs now which effectively violates rule 2.
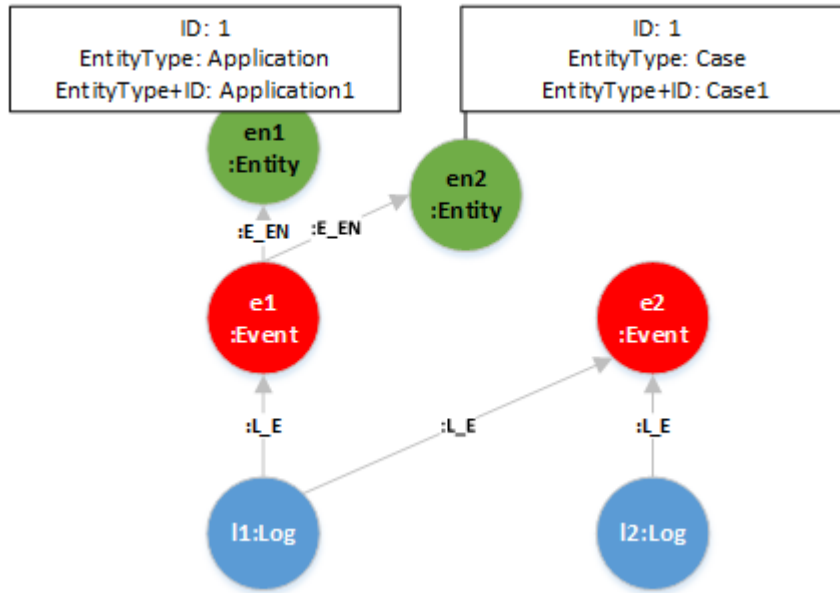
Figure 6.3: Core Pattern Non-Conform Instance

6.2 Directly Follows

Directly follows in process mining is the relation of two events a and b that consecutively follow one another in the context of a case. So if b happens after a and there is no other event in between them, we have the relation a directly follows b. Important here is the case context. Traditionally, event logs for process mining are focused on a single case identifier. The directly follows relation between events is defined in the context of an entity. Entities are described in section 2.2.1 and basically serve as sub cases that can, if the data allows correlation between entities, be flexibly be combined to a composite entity. Such a composite entity, for example as shown in our running example in section 4.1 can then serve as new context for directly follows relations of events. We define DF by the sequential order the events got recorded, i.e. event c directly follows event b if c happens to be recorded in the next row after b even if their timestamp is exactly the same. In contrast, partial orders treat events with the same timestamp as parallel events. For example events b and c directly follow event a and are both directly followed by event d. The patterns for the two DF options might look equal, but the rules would be quite different. For our definition of directly follows we require an explicit order of events per entity. As discussed in section 6.1, timestamps are an intuitive choice when it comes to the order of events. However, timestamps of events are often unreliable, e.g. because of their low granularity (containing only date information and no time information), so in practice we cannot always rely on it to reconstruct the order in a graph when only importing events as nodes with a timestamp and thus losing the actual sequential order of the source table. However, since the source of the event data are event logs, we cannot investigate the actual order in the source system and thus assume that the order they are recorded in the event log is the actual order in which the events occurred.

Pattern: Figure 6.4 shows the visual representation of the DF pattern.

Even though we could only depict an event node type and the *DF* relationship to represent the newly introduced elements of *DF*. There is no longer a single given case, but entities that give the context to *DF* relationships. This means there are rules defined which add constraints to instance level elements that have the node type *Entity* and thus, the *Entity* pattern node type is included. The *EntityTypes* relationship property plays also a key role with respect to the entity context of a *DF* relationship, i.e. two events a and b can have a *DF* relationship for multiple entities.

As discussed earlier, we decided to limit the elements of the pattern's visual representations
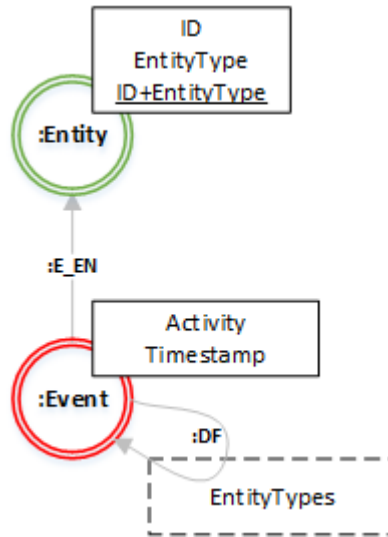
Figure 6.4: Directly Follows Pattern

to the minimum, to maximize the readability. Meaning we only include those elements that are somewhat affected by the pattern or important to understand the pattern even if the pattern inherits many elements from another pattern. This can either be a new node type or relationship type added by the pattern, or element types whose instance level elements are affected by a rule that comes with the pattern, e.g. by adding a new cardinality constraint. This should become clear with the *DF* pattern.

The events 4 and 5 of running example in table 4.1 would, in our entity concept, have a *DF* relationship for case and offer 2, the events 3 and 6 have a *DF* relationship for offer 1 and case, and so on. This is the kind of reasoning behind the visual representation of a pattern. You will find more elaborated examples at the end of this section. The pattern consists of all elements either defined in the pattern itself or inherited by other patterns. The newly introduced elements are marked as **bold** for better understanding. The full pattern definition looks as follows:

```
1  1_df = (
2      {//element types
3              Event {Activity, Timestamp}
4              Entity {ID, EntityType, ID+EntityType},
5              Log {ID},
6              E_EN {},
7              L_E {},
8              DF EntityTypes
9      }
10     {//node types
11             (:Event), (:Entity), (:Log)
12     }
13     {//relationship types
14             (:Event)-[:E_EN]->(:Entity),
15             (:Log)-[:L_E]->(:Event),
16             (:Event)-[:DF]->(:Event)
17     }
18     {//inherited patterns
19             0_core
20     }
21 )
```

Listing 6.2: DF Pattern Definition

For the written definition of the *DF* pattern, there is actually not so much difference to the core pattern. We have only one new element type that is materialized as relationship type *:DF*

with event node type as source and destination and the property type *EntityTypes* we already described. Everything else is inherited from the core pattern. The real difference of this and also of the following patterns, lies in the rules that come with the patterns. Also the newly introduced pattern *DF* relationship type can be mapped to the relationship's instances in the graph log:

The 'directly follows' relationships, DF, $\mathcal{R}_{df} = \{r | r \in \mathcal{R} \wedge "DF" \in labels(r)\}$, mapping to the $(: Event) - [: DF] - > (: Event)$ pattern relationship type. All inherited element types from the core pattern keep their mapping, unless a rule defines something else.

For the *DF* pattern we defined the following rules:

Rules:

1. A *DF* relationship can only exist between two distinct events e_1 and e_2.

2. All events that have a *DF* relationship with each other must each have a E_EN relationship to the same entity node: $(e_2$:Event)-[:E_EN]->$(en_b$:Entity)<-[:E_EN]-$(e_1$:Event).

3. The *Timestamp* property of the events e_1 and e_2 in a relationship $(e_1) - [: DF] - > (e_2)$ defines the direction of the relationship by satisfying $e_1.Timestamp \leq e_2.Timestamp$.

4. DF {EntityTypes} contains a list of (distinct) business entity types (en_b) for which the DF relationship of e_1 and e_2 holds.

5. There exists no e_x such that $e_1.Timestamp < e_x.Timestamp < e_2.Timestamp$ for any of the entity nodes this specific DF relation holds for.

6. Events e_1 and e_2 must be in the same log.

Rule 1 is to ensure that the *DF* relationship only exists between two different events. If we would let the *DF* self loop on the event node type in figure 6.4 be the only constraint we would allow a directly follows relation with the same event node as source and destination on instance level. This would read event *a* directly follows event *a* and this does not make sense. Rule 2 enforces the entity context for two events that have a directly follows relation. As we elaborated earlier in this section already, every entity can serve as case identifier and in some cases, if their events can be correlated in a logical way, combinations of entities can be used to form a composite case. This makes it necessary to ensure events with a directly follows relation actually relate to the same entity. Rule 3 makes sure that the source of a *DF* relationship did not occur earlier than its destination event node. Rule 4 defines that for every entity the *DF* relationship of the two events holds, a distinct entry with the corresponding entity type (en_b) must exist. Rule 5 formalizes that there may be no third event that can be ordered in between two events with a *DF* relationship based on its timestamp. Please note that timestamp as such is not bound on a strict format such as datetime. It is rather a property of an event node that is concerned with the temporal ordering of events of an entity. This means there could also be an index or any other concept suitable to create an order instead of actual time information. The last rule, number 5, defines that events with a *DF* relationship must be from the same log. This rule might seem superfluous at the first glance, because yet event logs mainly consist of one log with one case. In our framework, however, we are able to Handel multiple logs in a graph instance. This in turn enables us to correlate events and entities of interacting processes and mine the data on those processes from different angles. The following patterns will go deeper into the area of correlating events and entities, but they focus on events or entities that coincide with each other.

Examples: For the directly follows pattern we show two examples with subgraphs of a graph event log instance that conform to the pattern rules. Figure 6.5 shows a *:DF* relationship of the events *e1* and *e2* with respect to entity *en* with *EntityType* "Application". Rule 1 is followed, because the *:DF* relationship is not a self-loop as we have two distinct event nodes. Both events have a *:E_EN* relationship to the same business entity *en* which fulfills the requirements of rule
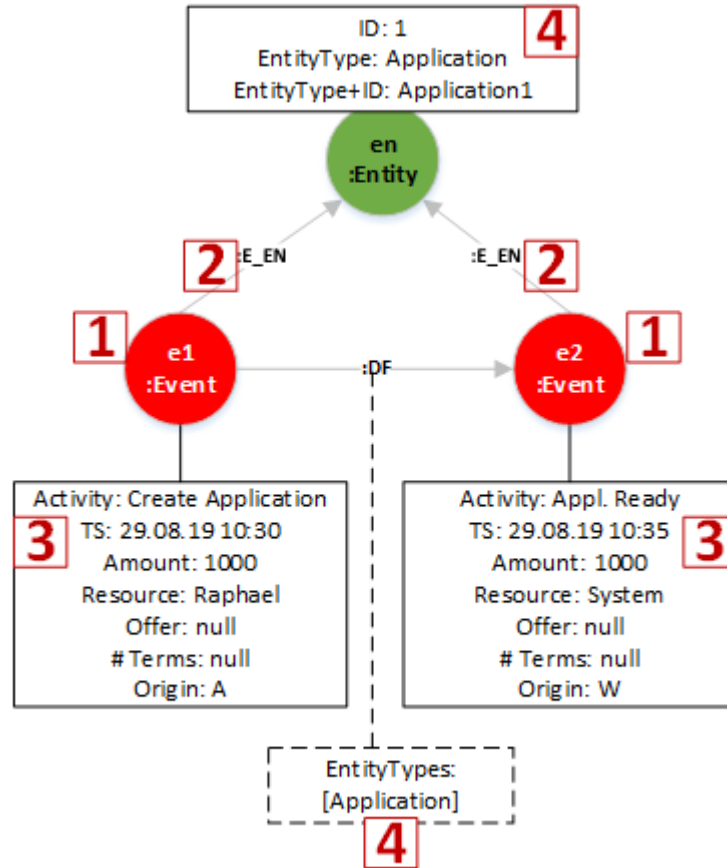
Figure 6.5: Directly Follows Pattern Rules 1-4 Example

2. For rule 3 we need to make sure that there is no event related to *en* that actually happened between *e1* and *e2*. In this example not other events exist and thus *e2* directly follows *e1* w.r.t. *en*. The *EntityTypes* property of the *:DF* relationship contains the *EntityType* of *en* which fulfills rule 4.

Figure 6.6 shows a similar scenario like the example above. We extended it with a log *l*.

Both events *e1* and *e2* are part of the same log *l*, which satisfies rule 6. This rule is necessary, because we do not expect events recorded in different logs to belong to the same process related context. It might, however, be the case that different logs contain events that coincide, i.e. originate from the same task execution, and thus might indeed have a relationship to events of another log. For our graph model we decided to keep those events apart from each other, instead of merging coinciding events to one node, because the identification of coinciding events usually requires advanced domain knowledge which should not be required in our scenario. The following patterns embrace exactly this subject.

The directly follows relation as we used it for this pattern is a commonly used way to define DF, i.e. we generally assume that the order in which the events are recorded in the event log is the same order in which the occurred in the information system, despite of potentially equal timestamps with too little granularity in the timestamp column such as date. However, the pattern approach could be extended to define other versions of the DF pattern. Even though we do not generally restrict how patterns can be used together or inherit each other, it should be obvious that the definition of a second DF pattern, e.g. to create partial orders, should involve some mutual exclusivity between the two patterns.
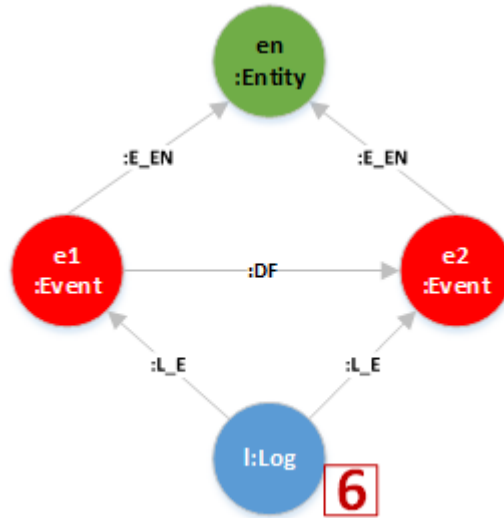
Figure 6.6: Directly Follows Pattern Rule 6 Example

6.3 Coinciding Events

If two events e^{l1} and e^{l2} coincide, they can be correlated by $(e^{l1}) - [e\_coincide] -> (e')$ and $(e^{l2}) - [e\_coincide] -> (e')$ relationships. This relationship can be constructed by using domain knowledge and can be applied to any number of events of the same log or different logs within a graph instance. One (e') could be used to correlate many different events. With the help of this pattern we can group any number of events that originate from the execution of the same task and thus actually are multiple materializations of the same event. Sometimes event logs contain events with two timestamps, one for the start and one for the end. In traditional sequential event logs the analyst usually has to decide which of the timestamps is used to determine the correct order of events at the time of loading the event log. Another approach to deal with events with start- and end-timestamp could be to treat them as two separate events, but this way most of today's process mining techniques would not deliver useful results on this data since they cannot refer to the coinciding events as the same activity. With the help of this pattern we can actually correlate events of a log that refer to the same activity and query the data as we desire, for example by using only one start or end event by a simple graph query with no need to create a new log or sort the log again, or by using the time interval between start and end events to determine whether events that appear to happen in sequence in the log actually happen in parallel. An other useful application of this pattern is to correlate events from different logs, for example events of batch jobs that affect different processes and in consequence lead to multiple events in the logs of these processes actually representing the same event. If we have enough domain knowledge to correlated these coinciding events from the data, we can use this pattern to incorporate this correlation information in the graph and derive further knowledge from it, such as interdependent entities for example.

Pattern: The definition of the coinciding events pattern looks as follows:

```
1 l_e_coincide = (
2     {//element types
3          Event {Activity, Timestamp}
4          Entity {ID, EntityType, ID+EntityType},
5          Log {ID},
6          E_EN {},
7          L_E {},
8          E_COINCIDE
9     }
```

```
10      {//node types
11              (:Event), (:Entity), (:Log)
12      }
13      {//relationship types
14              (:Event)-[:E_EN]->(:Entity),
15              (:Log)-[:L_E]->(:Event),
16              (:Event)-[:E_COINCIDE]->(:Event)
17      }
18      {//inherited patterns
19              0_core
20      }
21  )
```

<div align="center">Listing 6.3: E_COINCIDE Pattern Definition</div>

Similar to the *:DF* pattern, the *:E_COINCIDE*, we have only a small extension (bold) of the inherited elements. This also a good example to show why we need rules, because we actually add a very similar component to the graph as we did in the DF pattern, i.e. a relationship with *:Event* nodes as start and destination. The only difference, except for the rules, is the label (type) *:E_COINCIDE*. The effect on the graph instance, however, is completely different from the *DF* pattern. You can immediately see in figure 6.7 that the focus of the pattern is very different.
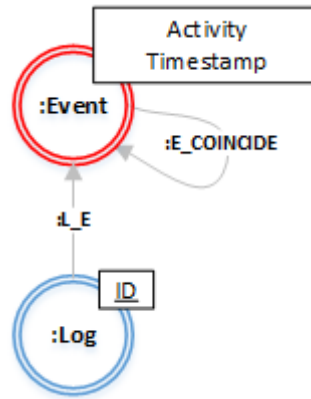
<div align="center">Figure 6.7: Events Coincide Pattern</div>

By comparing figure 6.7 and figure 6.4 it becomes also clear that the pattern elements are not the only determining factor for what's included in the visual pattern representation or what a pattern does. We added very similar element types to the pattern, but the rules affect the inherited components in a way that both figures look quite differently. The next step is to introduce the rules for the pattern:

Rules:

1. Events e_1, e_2 ... e_n can be correlated to each other through some new event node e', a meta or collector event that serves as single destination node for all *E_COINCIDE* relationships of the coinciding events.

2. Collector event nodes e' cannot be the destination of a L_E relationship: $|(: Log)-[: L\_E]->(e')| = 0$

3. Collector event nodes e' cannot be the source of any: $|(e' : Event)-->()| = 0$

4. e' correlates at least two event nodes: $|(:Event)-[:E\_COINCIDE]->(e':Event)| \geq 2$

5. Each regular event e can only be the source of one *E_COINCIDE* relationship: $|(e:Event)-[:E\_COINCIDE]->(e':Event)| \leq 1$

Rule 1 basically introduces a meta event e' which serves as destination for the *:E_COINCIDE* relationships of a set of coinciding nodes. e' is also the reason for the 'special cases' of events mentioned in section 6.1 when we introduced the core pattern, because we effectively need to alter the lower bound of the relations between event and entity and between log and event. Rule 2 regulates that collector nodes may not be part of any log since it is supposed to represent some meta information and it shall not obscure the information of the original event data. With rule 3 we effectively overwrite rule 4 of the core pattern and the lower bound of the cardinality of the event:entity relation changes to [1..n]:[0..n], i.e. every entity still requires at least one event, but an event can, as a collector event, can exist without a relationship to an entity node, but only under these specific circumstances. Rule 4 defines that at least two events must be correlated by a collector node, a rather intuitive rule. Rule 5 defines that a specific event cannot be correlated through multiple different collector nodes. In this thesis we solely use ready made event logs for our case studies. This means that hardly can determine which events actually coincide and which ones don't. To evaluate this pattern, we actually prepared one of the data sets to contain two events per one event in the log, one for the start timestamp and one for the end timestamp by keeping the rest of the attributes for both. Even though this is a rather artificial situation for such a pattern, it serves the purpose of feasibility evaluation. The real value of such a pattern, however, probably lies in the application on data that directly originates from a relational data source where one can better determine if events that 'appear' to affect different processes actually coincide. We want to emphasise once more that these patterns are just examples and there are many ways to handle such a structure. Another approach to handle coinciding events may be to combine them in a single node, which in turn implies significant changes in other patterns as well and especially to their rules. Changing the rules in turn significantly changes how we can query the graph and what assumptions are made at the time of the graph creation. For example, if the coinciding events would be incorporated as a single event node, which is how one might intuitively design it in a property graph, we would also need to change the rules of the core pattern to enable an event node to be related to multiple log nodes.

Examples: The simple example in figure 6.8 shows two events *e1* and *e2* of the same log that are correlated by a collector event *coll*.
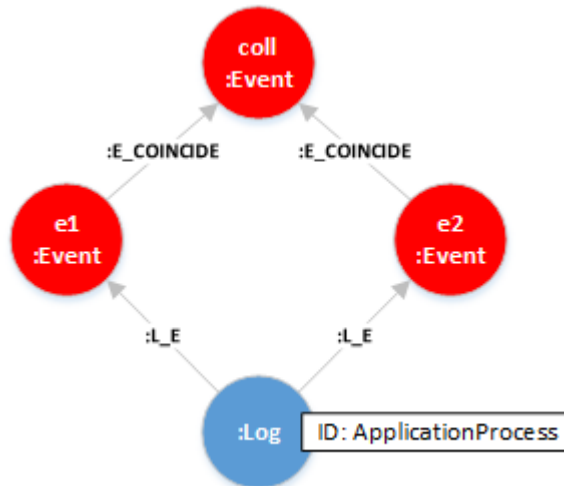
Figure 6.8: Coinciding Events Pattern Conform Instance

This is the pattern that we intended to model by the pattern and the rules. We wanted to correlate events of a log that have been created from the exact same task execution in the process. In this example, *e1* and *e2* belong to the same log, which is in line with the pattern definition and rules.

Figure 6.9 shows an example that violates the rules of the pattern.
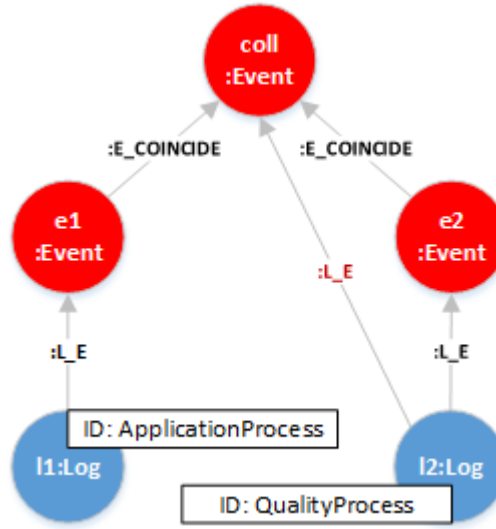
Figure 6.9: Coinciding Events Pattern Non-Conform Instance

In the example that does not conform to the pattern rules, we introduced a second log to illustrate in what situations events from different logs may coincide. Here we added a second log *l2*, a fictional log to complement the running example log *l1*. Say *l2* is a quality management process that is supported by the same information system as the application process and thus uses the same set of entities, such as *Applications* and *Offers* and thus, we have events that affect both processes at the same time. A *Offer Returned* activity in the application process might, for example, create an event relevant to both processes. In the example we show how these events can be correlated. The second log, however, is not what violated a pattern rule. The actual violation is the *:L_E* relation between the log *l2* and the collector event *coll*. Rule 2 defines that collector events may not be the destination of a *:L_E* relation.

6.4 Coinciding Entities

Similar to events, entities of different logs can also coincide. The same logic for the graph structure applies here. By correlating entities from different logs with each other in the graph event log, we enable the process analyst to correlate events from different processes that affect the same entities and thus might influence each other. Early versions of the coinciding patterns we developed and tested in our case studies have been more generic, such that the two patterns were one big pattern with the elements and rules of the two combined. In the end we decided to split them because we wanted to stay as flexible as possible, i.e. not every event log with coinciding entities also has coinciding events and vice versa. Every time we add an element type to a pattern of a graph it automatically becomes an element type of the schema, which is not what we want if the graph instance does not have any elements of that element type. Thus we again only added a few elements to the core as you can see in the listing below.

Pattern:

```
1  l_en_coincide = (
2      {//element types
3              Event {Activity, Timestamp}
4          Entity {ID, EntityType, ID+EntityType},
5          Log {ID},
6          E_EN {},
7          L_E {}
```

```
 8              EN_COINCIDE
 9        }
10        {//node types
11              (:Event), (:Entity), (:Log)
12        }
13        {//relationship types
14              (:Event)-[:E_EN]->(:Entity),
15              (:Log)-[:L_E]->(:Event),
16              (:Entity)-[:EN_COINCIDE]->(:Entity)
17        }
18        {//inherited patterns
19              0_core
20        }
21  )
```

Listing 6.4: EN_COINCIDE Pattern Definition

As the entity counterpart of *:E_COINCIDE*, *:EN_COINCIDE* actually builds the same structure with entities as *:E_COINCIDE* does with events. Thus the *:EN_COINCIDE* relationship has entity nodes as source and destination. The pattern to link coinciding entities becomes especially interesting in situations where multiple logs exists that share the same entity types. For example in many modern IT landscapes different processes such as change management and incident management relate to the same entity types such as hardware or software from very different angles. If you add additional processes such as an IT support process where support personnel interacts with users for different purposes like changes or incidents, we have very intertwined system of entities with varying influences from different processes. The event data that gets collected from this kind of environments would nowadays most likely be split into at least one event log per process which would be mined for insights in isolation. This pattern is designed to help redrawing those 'lost' connections and take them into account when analysing the data of the different logs. This is again motivated by the fact that we use ready made event logs as base. Event data directly extracted from relational sources might use a different approach, similarly to the approach discussed for coinciding events and with similar consequences. This means that coinciding entity nodes could also be represented as single nodes that then would be part of multiple logs. Figure 6.4 shows the visual representation of the pattern.

As we can see in the figure, the graph now also includes the entity node types and the *:EN_COINCIDE* relationship type as selfloop of the entity types. Even though it is not obvious, the differences between events and entities can be very large. In our concrete pattern collection we used here, entities have an *EntityType* pattern for example, which determines what entities can actually coincide with each other and which ones cannot. For events we don't have such a differentiation. This kind of differences lead to many different rules to cover the different attributes of the elements and certain combinations thereof we want to regulate in the actual graph instance. This is the second reason why we decided to split the patterns into two. As we already mentioned, per our definition entities are to some extend a more complex concept than events and consequently the number of rules is a bit higher.

Rules:

1. Two entities of the same entity type, i.e. en_1, en_2 such that $en_1.EntityType = en_2.EntityType$, can correlated with each other.

2. Similar to coinciding events, entities $en_1, en_2 ... en_n$ can be correlated with each other through some entity node en', collector entity that serves as single destination node for all *EN_COINCIDE* relationships of the coinciding entities, e.g.:
   $(en_1$:Entity$)$-[:EN_CONINCIDE]->$(en'$:Entity$)$<-[:EN_CONINCIDE]-$(en_2$:Entity$)$

3. The two entities cannot be part of the same log log_x: $\neg(en_1 \in log_x \land en_2 \in log_x)$

4. en' has no outgoing edges: $|(en') - - > ()| = 0$
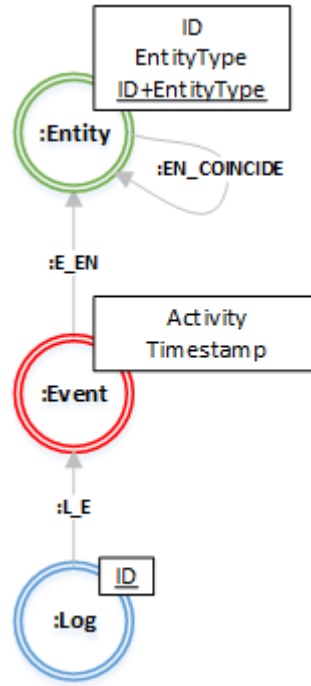
Figure 6.10: Entities Coincide Pattern

5. en' correlates at least two entity nodes: $|(:Event)-[:EN\_COINCIDE]->(en':Entity)| \geq 2$

6. Every entity en can only be the source of one *EN_COINCIDE* relationship:
   $|(en:Entity)-[:EN\_COINCIDE]->(en':Entity)| \leq 1$.

Rule 1 of the pattern for coinciding entities specifies that entities must, to be able to correlate them, have the same *EntityType*. *EntityType* is not to confuse with the graph element type we specify for a pattern or schema. *EntityType* is the attribute on instance level containing the information of the actual type of process entity, e.g. resources, documents etc. The rule ensures that we don't relate to a user and an invoice as coinciding entities. We intentionally do not enforce the equality of ID values, because it is possible that coinciding entities in different logs may have different IDs, e.g. by log prefix. We leave this open to the user and leave the ID matching to domain knowledge. In rule 2 we define that two or more entities can be correlated with each other through one entity collector. The collector node has the same properties as regular entity nodes it 'collects'. Thus, during the an implementation, the event log designer must ensure that the properties with unique flag also stay unique for the collector nodes as well. This could for example be done by adding a log-specific prefix to the regular entity nodes, but this is not regulated by our pattern rules as different analyses may require different implementations. Rule 3 defines that two coinciding entities cannot come from the same log. The reason for this is again that we use ready made event logs again. We simply lack the process knowledge if a sub process of case a coincides with a sub process of case b, so we make the assumption that all entities of one log are discrete entities. In rule 4 we again create an exception to a rule of the core pattern, because we define that entities without incoming *:E_EN* relationships can exist, or more specifically: we define that entity collector nodes can't serve as source node for any relationship. Thus we alter the lower bond of the event:entity relation to [0..n]:[1..n] if we only use the *:EN_COINCIDE* pattern on a log. In the case we use both patterns, *:EN_COINCIDE* and *:E_COINCIDE*, in one log we actually create a [0..n]:[0..n] event:entity relation, but of course only for the specific cases defined in the pattern rules. Rule 5 says we need at least two entity nodes to correlate, otherwise we can't use the collector entity. With rule 7 we specify that a regular entity cannot be linked to more than

one collector node, because this could lead to confusion when querying the coinciding entities over multiple paths.

Examples: Figure 6.11 shows how such a collector entity can look like. With the entity nodes
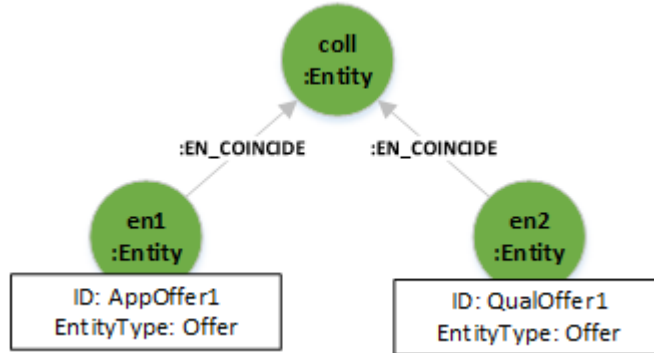
Figure 6.11: Coinciding Entities Pattern Conform Instance

en1 and *en2* of the same *EntityType* (rule 1) are correlated through a collector node *coll*. This example illustrates that the two entities do not necessarily have the same ID. Say the offer entity *en1* belongs to the log of our example process and the offer entity of *en2* refers to the same business entity, but from the perspective of the quality management process of the bank that reviews offers and thus is captured in a separate event log. With the *:EN_COINCIDE* pattern we are able relate these entities. Figure 6.12 shows a sub-graph that does not fully comply to the pattern's rules.
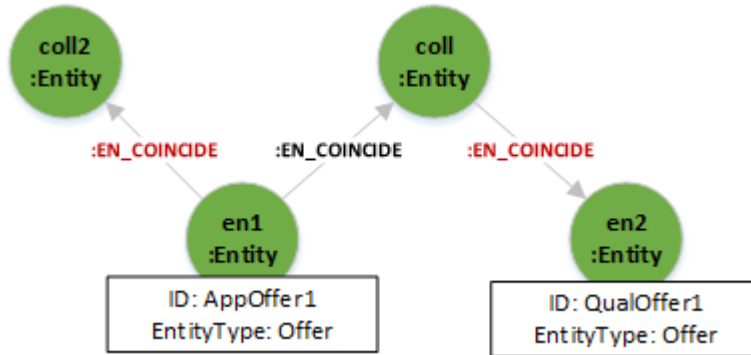
Figure 6.12: Coinciding Entities Pattern Non-Conform Instance

en1 got a relationship to a second collector node, which is a violation of rule 6. The second graph also got the direction of a *:EN_COINCIDE* relationship switched, making the collector node *coll* the source of this relationship.

6.5 Handover of Work

A handover of work social network is a higher level concept from the area of organizational process mining, as described in [27]. Handover of work, as the name suggests, is based on work handovers between resources in a process. To be able to create such a network, the log must contain resource information. In our framework, with the patterns we introduced already, resources are treated just as any other type of entity. This means, if the original log contained the information we need for *HOW*, it is already encoded in the graph and we just need to use it for our purpose. A handover of work relation is closely related to the directly follows relation of events. Say resources *r1* and

r2 carry out two subsequent tasks in one process instance leading to events *a* and *b*, i.e. *r1* is related to event *a*, *r2* is related to event *b* and *b* directly follows *a*. From that *DF* relationship we can derive the *HOW* relationship, *r1* hands over work to *r1*. This explains the concept on which we based the design of the *HOW* pattern. The *HOW* pattern is the first pattern to inherit two other patterns, the core and the directly follows pattern. The pattern definition can be found in the listing below. The element we added here is the *HOW* relationship with its *EntityTypes* property. While the *HOW* relationship implements the handover concept as described earlier, the *EntityTypes* property actually mirrors its counterpart from the *DF* relationship from which we derive the *HOW* pattern. This *EntityTypes* relationship property of *HOW* is used to 'mirror' the *DF* relationship's *EntityTypes* property it has been derived from. By following this logic we can construct the *HOW* relations and mine a *HOW* social network, a social network containing all hand overs between resources of a process, per entity. Moreover, we can basically specify any entity as resource and, based on that, create a *HOW* social network. In event logs with users, user groups and machines for example, we can freely choose which entities we want to consider for organizational mining. This makes the pattern to one of best examples how suitable property graphs are to store multidimensional event data.

    Pattern: The listing below shows the definition of the *HOW* pattern.

```
 1  2_how = (
 2      {//element types
 3            Event {Activity, Timestamp}
 4            Entity {ID, EntityType, ID+EntityType},
 5            Log {ID},
 6            E_EN {},
 7            L_E {},
 8            DF {EntityTypes},
 9            HOW EntityTypes
10      }
11      {//node types
12            (:Event), (:Entity), (:Log)
13      }
14      {//relationship types
15            (:Event)-[:E_EN]->(:Entity),
16            (:Log)-[:L_E]->(:Event),
17            (:Event)-[:DF]->(:Event),
18            (:Entity)-[:HOW]->(:Entity)
19      }
20      {//inherited patterns
21              0_core, 1_df
22      }
23  )
```

Listing 6.5: HOW Pattern Definition

    As mentioned already, the *HOW* pattern is the first one to inherit two other patterns, *0_core* and *1_df*. Next to the inherited elements, there is again only one new element unique to this pattern, the *HOW* relationship with its *EntityTypes* property. The source and destination node types are entities. Figure 6.13 shows the visual representation of our handover of work pattern.

    The entity node type got a self loop. Without rules we now could create handovers between invoice entities and resource entities, which does not make sense. Interestingly, handovers between resources of different logs is not possible unless we allow it by a rule in this pattern even though the *Log* node type does not show in the visual representation in figure 6.13, meaning the pattern does not have a rule affecting this element type. How can this work then? The answer is: inherited rules. In rule 6 of the *DF* pattern we specified that the source and destination node of a *DF* relationship must be from the log and because we derive the *HOW* from the *DF* we ensured already that the *HOW* relation is based on the same log's events. The only question we still may ask is: how do we treat the resources? Should we treat them as distinct entities per log or should we use single nodes over all logs, or even a completely different representation? This is a similar discussion like we had for coinciding events and coinciding entities in previous patterns and the answer is: it depends. There is no definitive answer to this because different data and
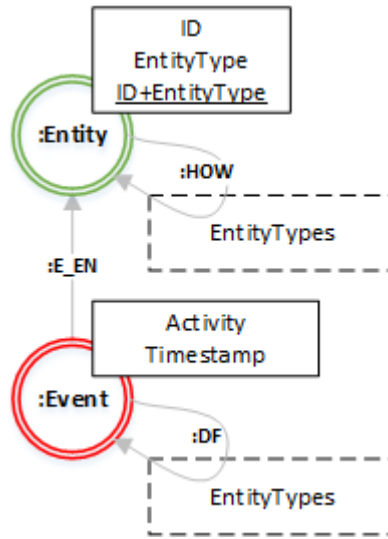
Figure 6.13: Handover of Work Pattern

analysis question require different solutions. Our proposed pattern approach deliberately leaves such design choices to the analyst. For our patterns, we already have implemented a solution with the *:EN_COINCIDE* pattern, since resources are as well entities and *:EN_COINCIDE* specifies that we have to treat each entity as as discrete node per log and correlate them by collector entity nodes. Please note that we also have the option to use the *HOW* pattern without *:EN_COINCIDE* for a graph instance, e.g. for data sets with only log only, data sets without coinciding entities, or by the help of a new pattern that treats coinciding entity nodes differently. However, we still need to define rules for our *HOW* pattern.

Rules:

1. Iff two events e_1 and e_2 have a DF relationship (e_1:Event)-[:DF]->(e_2:Event) with the related resources (entities) en_1 and en_2:
   (e_1:Event)-[:E_EN]->(en_1:Entity) and (e_2:Event)-[:E_EN]->(en_2:Entity)
   then the two resources can have the relationship (en_1:Entity)-[:HOW]->(en_2:Entity)

2. en_1 and en_2 must have the same entity type : $en_1.EntityType = en_2.EntityType$

3. HOW relationships have a property *EntityTypes*, a list of entity types for which business entity (en_b) the HOW relationship is valid, i.e. for the relationships *df* and *how* with:
   (en_1:Entity)<-[:E_EN]-(e_1:Event)-[df:DF]->(e_2:Event)-[:E_EN]->(en_2:Entity) and
   (e_2:Event)-[:E_EN]->(en_b:Entity)<-[:E_EN]-(e_1:Event) and
   (en_1:Entity)-[how:HOW]->(en_2:Entity),
   then en_b.EntityType $\in$ df.EntityTypes and en_b.EntityType $\in$ how.EntityTypes.

Due to the inherited rules from the core and df patterns we can keep the number of dedicated rules for *HOW* relatively low. Rule 1 formalizes the handover of work concept as described earlier in this paragraph in terms of how the handover of work can be derived from the directly follows relations. With rule 2 we ensure that handovers can only happen between entities of the same type, e.g. employee to employee or machine to machine. Since handovers between resources can be based on events of multiple entities, the *EntityTypes* list of the *DF* relationships and the *EntityTypes* list of the *HOW* relationship must always both include the *EntityType* of the business entity en_b. To conclude the *HOW* pattern we want to show some examples.

Examples:

We want to illustrate the handover of work pattern with some examples. Figure 6.14 shows a partial graph instance from our running example on which we find all rules and the relevant parts of the pattern. Note that we use the Cypher annotation to assign variables to certain elements, e.g. (en1:Entity) means that en1 is of type :Entity and refers to the entity node with the properties shown in the properties box. Since we are looking at a instance and not at a schema, we cannot
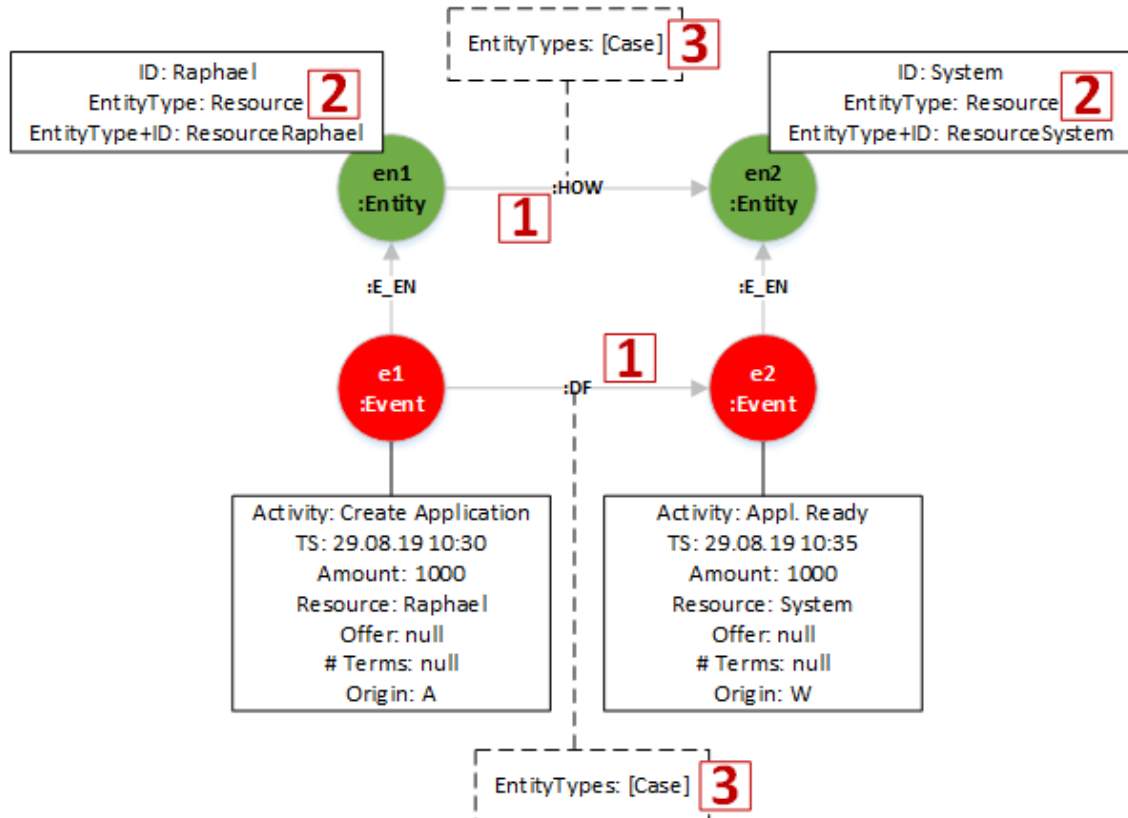
Figure 6.14: Handover of Work Pattern Conform Instance

reason whether the patterns are fully included in the graph's schema. What we can do, however, is assess the instance's conformance to the rules. The red boxes with number 1 inside show the graph elements that are affected by rule 1, i.e. the two events *e1* and *e2*, which are related to *en1* and *en2* respectively, have a *:DF* relationship. Thus, the requirements for a *:HOW* relationship between *en1* are met and rule 1 is satisfied. Rule 2 requires that *en1* and *en2* have the same *EntityType*. As we can see in the property boxes of *en1* and *en2*, rule 2 is satisfied as well. Rule 3 is to make sure the entity type of the actual business entity, i.e. not the resource entity, we want to create the handover of work for is also the entity type for which the *:DF* relationship is valid. This is determined by the elements of the list property *EntityTypes* of the *:DF* and *:HOW* relationships.

If we compare the above situation with figure 6.15, we can see that some elements have changed.

This figure shows a similar pattern of the graph instance that does not comply with rules 1 and 3. Rule 1 is violated because *e2* is not related to *en2* and rules 2 does is violated for this sub-graph, because the *:HOW* relationship has source and destination nodes of the same *EntityType*. Rule 3 in turn is violated, because *:HOW* and *:DF* refer to different business entities in their *EntityTypes* properties.
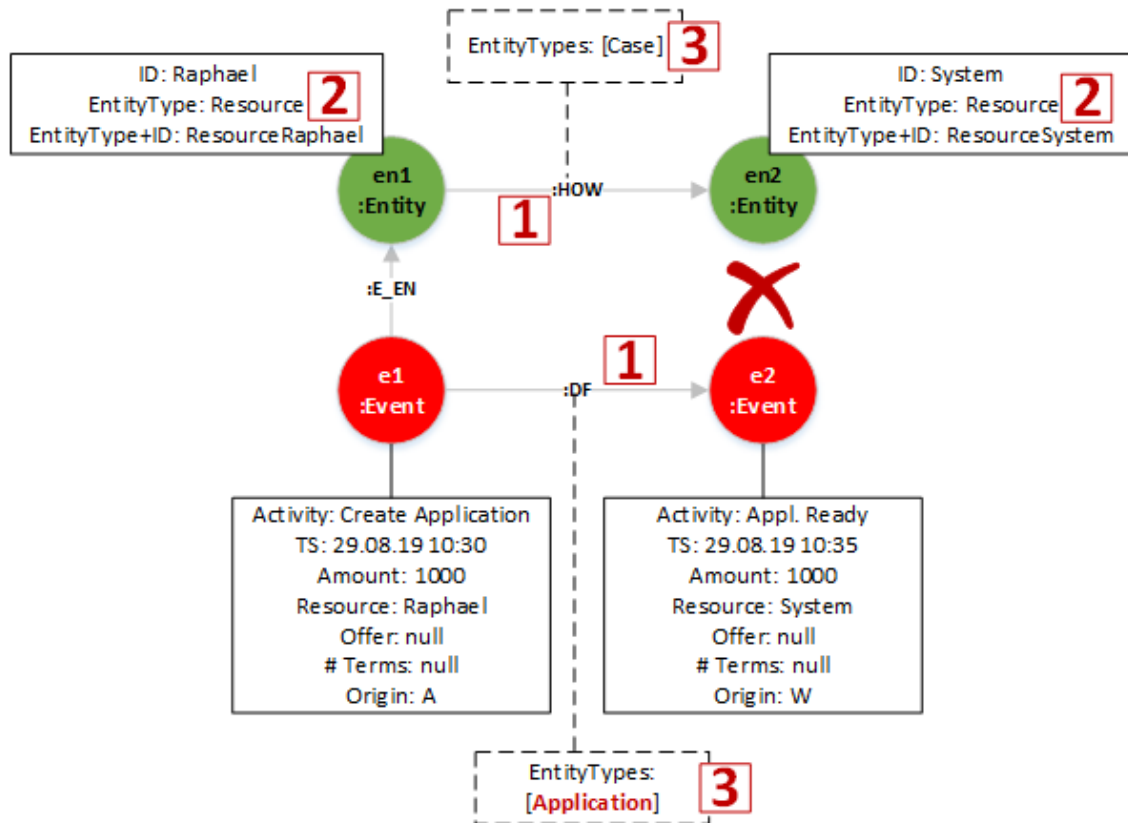
Figure 6.15: Handover of Work Pattern Non-Conform Instance

6.6 Schema Validation

With the patterns and rules introduced in the previous sections, we have everything that satisfies our requirements for a event graph schema. The only problem left is how to validate the schema and the instance, i.e. how to validate that all the "conforms to" dependencies in figure 5.5 are met?

In a first step we want to validate that a schema conforms to the defined patterns. A schema conforms to its patterns if it contains all element types, nodes types and property types from all its patterns. This means, if we merge all components of all the patterns for a schema, the result of the merger must be a subset of the set of components of the schema. We can validate this in different ways. For our case studies we closely inspected the resulting schema representation native to Neo4j and compared it to the patterns. Figure 4.2 shows such an output of Neo4j from another paper. By selecting the different components, we can inspect all the property types. A second way to validate "schema conforms to patterns" could be a dedicated graph instance that represents the graph schema and can be queried according to the patterns, i.e. define a dedicated query for every node type and relationship type from the patterns expecting exactly one match. For example, if a pattern defines the schema must contain *:Event* nodes, we define a query for exactly that node.

```
1 MATCH (e:Event)
2 RETURN e
```

Listing 6.6: Match :Event Nodes

Similarly, we can define a query for the *:DF* relationship in the schema graph.

```
1 MATCH (:Event)−[r:DF]−>(:Event)
2 RETURN r
```

Listing 6.7: Match :DF Relationships

If we run such a query for every node and relationship type and all return exactly one graph element, we are certain that the schema conforms to its patterns.

The second conformance we want to check is the one between the graph instance and the rules. Here we want to whether the constraints and cardinalities of defined by the rules are violated or not. In the case studies, we again followed a rather pragmatic approach and queried samples of affected patterns and visually checked their conformance. The affected parts on graph instance level are closely elaborated in the example sections of the respective patterns. A more systematic way to check the instance for rule conformance is to define queries that match patterns that are actually forbidden by rules. In the handover of work example in figure 6.14, rule 2 defines that the *EntityType* of the two entities handing over work must be equal. To test for conformance we can define a query on the graph instance:

```
1 MATCH (en1:Entity)−[:HOW]−>(en2:Entity)
2 WHERE en1.EntityType <> en2.EntityType
3 RETURN en1, en2
```

Listing 6.8: Query for wrong HOW Relationships

If this query returns a matching result, we are certain that the instance does not conform to the rules. For the rules we actually must test on non-conformance, because we restrict the graph instance, as opposed to the schema where we define mandatory components with a pattern. However, since we, in some patterns, add exceptions to other rules, these queries must be defined carefully.

The validation was not the focus of this research and developing a full, systematical validation methodology was out of scope. This, however, is a potential next step to enhance the pattern models in future research.

6.7 Summary

The overall goal of this thesis is to find a general way to encode multidimensional event data in a property graph. Due to the lack of a standard schema for property graphs, we could not use a out-of-the-box schema notation and apply it to our domain of interest. The development of such a standard is ongoing since years, indicating that this is not a trivial thing to develop. With the patterns in combination with the rules we found a way to overcome the lack of expressiveness of the schema presentation shown in Chapter 3. We furthermore introduced examples of how the framework can be applied to a specific domain, which is process event data in our case. We also proposed ways to validate conformance of the different framework concepts.

Chapter 7

Evaluation

In this chapter we show how the schema framework can be applied to different data sets with different kinds of multidimensional event data. In total, we conducted six case studies on five different event logs. Section 7.1 lays out the goal and the overall setup of our evaluation.

In section 7.2 we give an in-depth description of how the case studies have been executed, i.e. how we transformed the sequential source logs into multi-dimensional graph event data with a validated schema. We explicitly explain the executed steps on the example of BPIC 17 as representative procedure we used for all case studies. To conclude this chapter, we summarize the findings of the evaluation in section 6.7.

7.1 Evaluation Setup

We chose five different sequential event logs as basis for six case studies to demonstrate a possible way how the framework can be applied for these different multidimensional event data and have indeed been able to store these data in labeled property graphs with a corresponding schema. We used data sets of different BPI Challenges for the case studies. The BPI Challenges are academically driven process mining contests where participants deal with real live event data. These diverse, real live data sets form a suitable base to explore different forms of multidimensional data in event logs. The main disadvantage of using this data is that for all of these data sets, a preselection of the data and unknown design choices for the flat, sequential event logs have been made by third parties, so we had to rely on our own assumptions when it came to domain knowledge of the different data sets.

The objectives of the evaluation is to test our proposed LPG schema framework on real live event data by answering the following questions:

1. Can the schema definition proposed in Chapter 3 and the graph event concepts defined in Chapter 4 be applied to all data sets?

2. Are all schema patterns proposed in Chapter 5 applicable or do the patterns have to be adapted?

3. Can the entity concept be applied to the proposed solution?

4. Can meaningful queries be formulated on the data?

5. How is the performance compared to the sequential counter parts?

The main purpose of the case studies is to show that we can apply our graph schema framework to different data sets. They are explicitly not designed to show how process mining questions can be answered, this has been shown in previous studies [11, 12]. We did, however, adapt some process mining queries from these papers to our new schema structure to show that the same information can be retrieved from graph data based on our schema approach.

For all case studies we used the same machine, a 6 core Intel i7-9850H CPU @ 2.6 GHz with 32 GB of memory, and the same procedure of data preparation and graph creation to maintain some baseline of comparability. The development of the patterns has primarily been done on sample sets of the case studies in an iterative approach to find a sound set of patterns that can be used for all the data sets. The usage of resources and time for creating the respective graph instances have been recorded to be able to assess feasibility and usability of the proposed framework for the different data sets.

This thesis makes a proposal for writing down event graph schemata for event logs. It only provides the syntax for doing so while semantics of the schema (for schema validation) are derived for each concrete schema instance. As described in section 6.6, the schema validation has not been done systematically, but in a rather pragmatic way, i.e. by inspecting the Neo4j schema (similar to figure 4.2) of a graph instance and matching it with the defined schema. The instance's conformance to the rules of the it's local patterns has been checked by individual queries for (by rule) forbidden or prescribed data structures as shown in the example query in section ! 6.6. For every case study we show the data schema of the source event log and the assessment of it's multidimensionality, the data import, the creation of the log(s) and entity types as outlined in figure 7.1. Furthermore, as every event log has it's individual data structures, we individually show further graph event data concepts, such as coinciding events to tackle the data divergence problem, or creating case notions from base entities, for those some data where the source data schema supports the definitions of these graph concepts. To test and evaluate the resulting graph instances, custom Cypher queries and data profiling capabilities of Neo4j have been used to check if the validation requirements specified in section 6.6 are met. Additionally, we defined individual queries to demonstrate different aspects of multidimensional event data and explain how they are incorporated in our schema framework.

7.2 Execution

The case studies haven been carried out on the property graph database engine Neo4j with its native query language Cypher as introduced in section 2.5. The actual implementation has been done with Python and the py2neo package. This way we have been able to automate the graph implementation and optimized the iterative development of the patterns. Every graph instance requires a set of Cypher queries to import the data from the source log in CSV format and create the graph structure as defined in the respective schema and patterns. All scripts to the case studies can be found in appendix A. The scripts allow flexible switching between sample sets and the full data sets of the event logs to be used for a script execution. Identifiers for cases of each log are hard coded in the scripts to ensure comparability of the results.

Even though these scripts vary from case study to case study they all have a common process they follow. This process has been used to semi-automate the more than 100 iterations of creating and testing the different graph structures of the case studies with different configurations. The process as outlined in figure 7.1 includes:

First, the data import which consists of

- Analysis of the source data schema, i.e. what entities exist in the source event log, how are they identified and how do they relate to events and other entities?

- Data preparation, i.e. make the source log ready for import in Neo4j by formatting timestamps to a digestible format for example.

- Import the source log into a Neo4j database, i.e. create an event node for every row with properties for every attribute.

- Define the core event data schema, i.e. define unique constraints as defined in the global graph schema.

Even though XES format can directly be used by our Python scripts, e.g. by using the PM4Py package [3], we decided to define a common baseline over the case studies such that every input and output of the Python scripts is in CSV format. The output format was basically predetermined by Neo4j, since its import functions are mainly based on CSV formatted data. Standardizing also the input format to CSV was mainly motivated through keeping the dependencies of our Python scripts as minimal as possible to minimize potential barriers for later replication. Thus, in the first step we used the ProM Lite 1.2[1] process mining tool to convert the XES file to CSV format.

Importing the event nodes is done by the CSV import functionality native to Cypher and Neo4j. The import serves the purpose to transport all information from the CSV log to the graph such that every row from the CSV becomes an event node and every column becomes a property key to the events. The cell values become the property values.

Second, for every log in the data set we perform the tasks

- Create a log node with a unique ID property.

- Correlate this log's event nodes to the log node.

- Create the entity types that exist in the log. For every entity type we need to perform:

  - Create entity nodes with a unique ID property and a property indicating the entity type. If resource information exists, resource entities should be created first.

  - Correlate the events to the respective events to the entity nodes.

  - Create the temporal order relations between the events w.r.t. the entity type.

  - Optionally, if resource information is available, create the handover of work w.r.t. to the entity type.

Third, if the source data schema permits, we can choose to add additional data structures to the graph event log, such as

- Correlate coinciding events, i.e. tackle event data divergence.

- Correlate coinciding entities.

- Create derived entity types, i.e. use base entity types to create new "cases" of entities.

Figure 7.1 gives an overview of our graph event log process based on CSV source event logs.

The blue boxed tasks have been done for every graph event log in our case studies. The tasks in dark grey boxes are optional in a sense that not every event log had sufficient data to apply them. The bright grey boxes are analog to sub processes. The tasks in *Log* must be performed for every log and the tasks in *Entity Type* for every entity type accordingly. The output of *Entity Type* is multiple entity nodes per entity type and there can be multiple entity types per log. This process, however, is just the standard approach for every log. There may also follow different steps, e.g. defining a case from combined entities like we show in the BPIC 17 case study in section 7.3.

In the following paragraphs we introduce each case studies and the context of the event logs.

---

[1]http://promtools.org
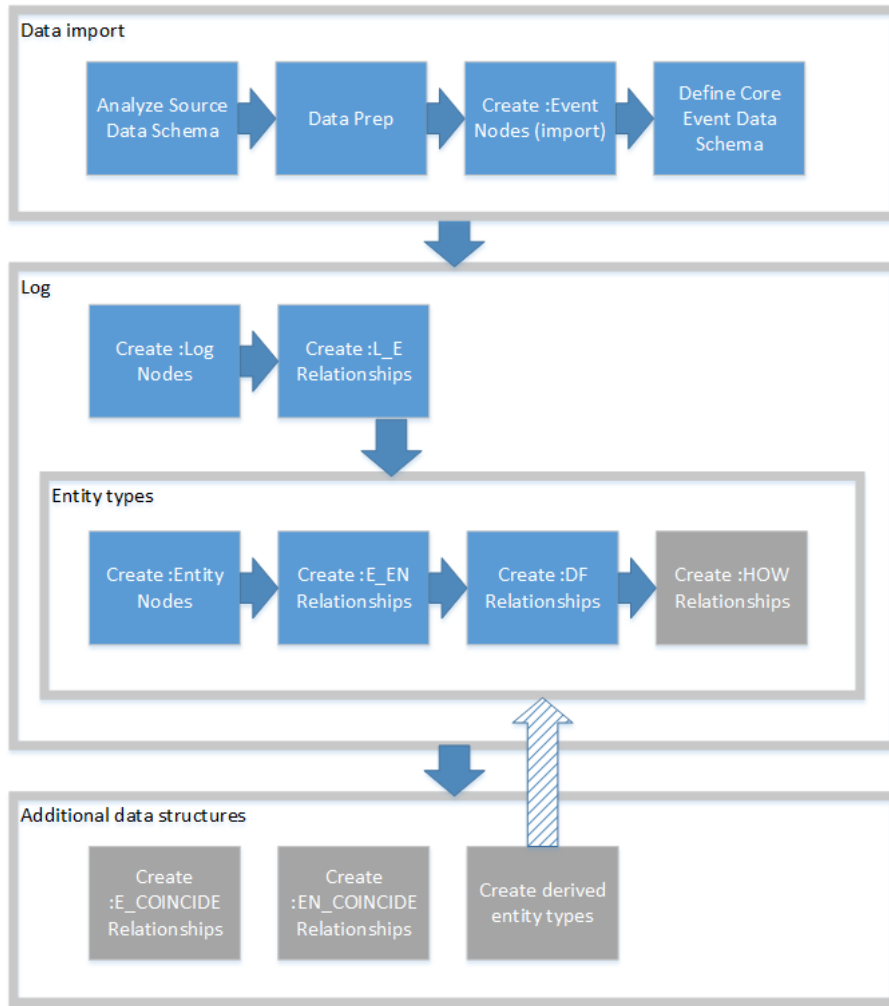
Figure 7.1: Create Graph Event Log Process Overview

7.3 Results on Loan Application Process

Therefore, the data should, to a certain extend, be familiar to the reader already. The loan application process is the event log of the BPI Challenge 17 [32]. It contains the data of a loan application process of an online system of a Dutch financial institute. A simplified version of this data set has been used as running example throughout the thesis.

Event Data Size: The log contains 561,671 unique events and 31,509 cases.

Activities: The events include the 26 different activities that are listed in table 7.1 in the left column.

Entities: Next to the defined *Case*, we identified the *Application*, *Workflow* and *Offer* as entities of the business process. The event data also includes resource information that, as per our definition can also be considered an entity. From the business process perspective, however, resources are rather complementary entities, i.e. they conduct activities, but do not have dedicated activities in the process itself. From the graph perspective, we are primarily interested in events and how they correlate to the different entities.

Activity-Entity-Relations: Thanks to the activity naming and an event attribute indicating the "EventOrigin", we are able to correlate events with their business entities without relying on external business knowledge. We show the full activity-entity mapping in table 7.1.

Activity	Entity
A_Create Application	Application
A_Submitted	Application
A_Concept	Application
W_Complete application	Workflow
A_Accepted	Application
O_Create Offer	Offer
O_Created	Offer
O_Sent (mail and online)	Offer
W_Call after offers	Workflow
A_Complete	Application
W_Validate application	Workflow
A_Validating	Application
O_Returned	Offer
W_Call incomplete files	Workflow
A_Incomplete	Application
O_Accepted	Offer
A_Pending	Application
A_Denied	Application
O_Refused	Offer
O_Cancelled	Offer
W_Handle leads	Workflow
A_Cancelled	Application
O_Sent (online only)	Offer
W_Assess potential fraud	Workflow
W_Personal Loan collection	Workflow
W_Shortened completion	Workflow

Table 7.1: BPIC 17 Activities per Entity

Events per Entity: We can divide the 561,671 events into 239,595 application events, 128,227 workflow events and 193,840 offer events. For resources it does not make sense to count dedicated events, since resources are rather complementary entities to the events and thus don't represent their own process. Nonetheless, we can in turn treat resource entities as case to retrieve the sequence of events associated to the resources. The log contains 145 resources, i.e. 145 resource cases. Because of the 1:1 relation, application and workflow have with 31,509 and 31,500 an almost equal number of cases and offer has a 42,995 case count.

Attributes and Entity Identifiers: The BPI Challenge 17 event log has 19 columns which serve as attributes to either one of the entities or to events. Generally, every column with a suitable identifier can be used to create an entity, i.e. every column that doesn't contain unique values. How useful such an entity is can only be answered with at least a basic idea of the domain of the process. In the BPIC 17 data set, we have a case ID, which apparently is an identifier for the applications, because all values in this column start with "Application_" and end with a number unique to every case. Next to the cases, we have a offer ID with a structure similar to the case ID.

Case-Entity-Relations: A case effectively consists of a combination of the three entity types application, workflow and offer. Application and workflow have a 1:1 relation whereas application (and workflow respectively) to offer is a 1:n relation. However, in some cases, no workflow events exist for a case ID. We assume that these cases may simply be incomplete or otherwise faulty. In table 7.2 we categorized every column to the best of our knowledge and as we are no domain experts able to verify these assumptions, we define our assumptions as given. For our research problem, it is not necessary to have a perfect representation of the respective domain. The identification and definition of the multidimensionality of the data is key. Thus, like for the previous case studies, we assessed every column whether it can potentially be used as identifier for an entity. Candidates for such an identifier can be all ID fields that allow grouping such that multiple events can be associated with such IDs. Again, assumptions need to be made according to the relevance of such an entity, i.e. does it make sense to associate an entity with some kind of (sub) process? The *Action* column for example has a number of distinct values like "Created", "statechange" or "Obtained" for certain events and thus could potentially serve as entity identifier. In the context of the application process, however, we assume that the *Action* column provides some meta information e.g. from a workflow management system for the status of certain process tasks or entities and therefore should not be treated as its own entity. The *Offer ID* column on the other hand seems to exactly fit our expectations of an entity, i.e. individual entities are associated with multiple events and can logically be related to as sub process in the given log context. In section 4.1 we discussed this log's entities and their relations already so we do not need to elaborate on them too deep anymore.

Table 7.2 shows how we assigned the identifiers to entities. Since we assume that applications and workflows have a 1:1 relation, we can use the case ID column for both.

For this data as described and modeled above, we now conducted two analyses:

1. Repeat the analysis of [11] and [12] through the generic approach proposed in this work.

2. Investigate the ability to handle data convergence [20].

For 1, we used the data as is. For 2, we used the two event attributes *Start timestamp* and *End timestamp* to create a log where we certainly know what events coincide by 'splitting' the events to a start and end event. With this artificially generated situation we want to simulate situations where we actually have multiple events that originated from a single event in the source information system, also known as data convergence problem as described by Lu et al. in [20]. Please refer to section 2.2 for a detailed description of this type of problem in event data.

7.3.1 Graph Creation for BPIC 17

This case study uses the original log with two timestamps per event. In the following paragraphs we describe how we created the graph by following the process shown in figure 7.1.

Column	Example	Entity ID for	Attribute to
Case ID	Application_652823628	Application, Workflow	Application, Workflow
Activity	A_Submitted		Event
Start timestamp	2016/01/01 10:51:15.304		Event
End timestamp	2016/01/02 12:30:28.633		Event
Loan goal	Home improvement		Application
Application type	New credit		Application
Requested amount	15000		Application
Action	Created		Event
FirstWithdrawalAmount	500		Offer
# of terms	33		Offer
Accepted	True		Offer
Offer ID	Offer_148581083	Offer	Offer
Resource	User_38	Resource	Event
Monthly cost	200		Offer
Event origin	Offer		Event
Event ID	ApplState_752879093		Event
Selected	True		Offer
Credit score	979		Offer
Offered amount	15000		Offer

Table 7.2: BPIC 17 Data Description

Data Import

Data Preparation − This part describes how to get the data right. The log can be downloaded in XES format. With ProM Lite 1.2 we converted the log to CSV format. We identified unexpected data conversions for some attributes for BPIC17. Please refer to appendix A.2 for more information. After the conversion from XES to CSV, we have a log file with 1,160,405 rows with many duplicates. These are removed during the preparation. The data preparation also included the transformation of the timestamps to a string with datetime format according to the ISO 8601 standard. This allows us to directly import the timestamps as Neo4j datetime datatype. After the preparation steps we have a CSV file with 561,671 rows (events) and a size of 0.13 GB ready for import.

Event Node Import − This part is about the data import. After the log in CSV format has been prepared, we import the complete log into the graph database system. After 32 seconds, the import has finished and with a resulting graph database with 561,671 *:Event* nodes, 0 relationships and a size of 0.93 GB.

Unique Constraints − In this part describe how we created the unique constraints. In Neo4j, we do not require actual data objects to be present in order to define unique constraints for object types. To enforce the uniqueness of our entity identifiers, we define the unique constraints before we create the actual nodes. To follow our schema definition, we define a unique constraint on the property type "ID+EntityType" of the *:Entity* node type, which we renamed to *uID* in the implementation for practical reasons, since the name in the pattern shall indicate from what other properties this property is constructed. We also define unique constraints on the *ID* property type of the *:Log* node type, which is the last step of the data import section in our graph event log creation process.

Log

Create Log Nodes − The BPIC 17 event log data set consists of only one a single log, so there is exactly one *:Log* node to be created in the next step.

Create Log to Event Relationships − With that log node, we can now proceed and create the *:L_E* between this log node and all of the 561,671 *:Event* nodes, adding 561,671 *:L_E* relationships to our graph.

Entity Types − The sub tasks for entity types, as shown in the "Entity Types" box in figure 7.1, are performed in sequence and need to be executed for every individual entity type, i.e. we loop over the tasks for every entity type.

Entity Nodes −− As a first step, we create the entity nodes. As mentioned in the BPIC 17 log description above, we have identified *Application*, *Workflow* and *Offer* as process entities. Additionally, we have resource information we want to use for organizational event data structures and thus create a *:Resource* entity as well. This has been done by using the information encoded in the event properties. For every entity type, we have defined an entity identifier as shown in table 7.2. For every entity we want to create, we can use these identifiers to create entity nodes, i.e. for every unique value of the identifier one node is created. For creating resource entity nodes, we can use the query shown in listing 7.1.

```
1 MATCH (e:Event)
2 MERGE(r:Entity {ID: e.resource})
3 ON CREATE SET r.uID = ('Resource'+toString(e.resource)), r.EntityType = 'Resource'
```
Listing 7.1: Create Resource Entity Nodes (Cypher)

The query shows that we only match event nodes and use their properties, e.resource in this case, to create distinct *:Entity* nodes with the "MERGE" clause.

For the business entities in this data set, we also need to take the "EventOrigin" into consideration. We show this by example of the *Application* entities in the following query:

```
1 MATCH (e:Event)
2 WHERE e.EventOrigin = "Application"
3 WITH e.CaseID AS id
4 MERGE (en:Entity {ID:id, uID:("Application"+toString(id)), EntityType:"Application"})
```
Listing 7.2: Create Application Entity Nodes (Cypher)

In this query we actually set the property values and keys in the "MERGE" clause directly, instead of using an additional "ON CREATE SET" clause. In fact, there is no difference if the query is correctly defined. To identify our application entities, we combine the "EventOrigin" property with the case identifier. For the entity's unique identifier, we also use the "CaseID" with the entity type as prefix, to create uniqueness to the identifiers among the different entity types.

:E_EN Relationships −− Next, we correlate the entity nodes to the events. For every single entity we create *:E_EN* relationships between them and their respective events as shown in the example for resource entities in listing 7.3.

```
1 MATCH (e:Event)
2 MATCH (r:Entity {EntityType: "Resource"})
3 WHERE r.ID = e.resource CREATE (e)−[:E_EN]−>(r)
```
Listing 7.3: Create :E_EN Relationships (Cypher)

:DF Relationships −− Now we can go on to the *:DF* relationships. Once we have correlated the events with their entities, we can continue and create the entity specific *:DF* relationships. With the help of the original sequence of the events, we generate an entity specific index to be able to generate the *:DF* relationships over the events of the newly created entities. We used Python for that, firstly because the handling of graph objects that have no direct relationship yet was far more intuitive and secondly, we created the graphs with Python scripts anyway so this could be easily integrated. In the next query we used the pandas and py2neo packages in Python to create a sort order index for the newly created entity type.

```
1 query = 'MATCH p = (ev:Event) −[E_EN]−> (en:Entity {EntityType: "Application"})
       RETURN ev ORDER BY ev.case, ev.idx'
2 output = Graph.run(query).data()
3 entityIdx = 0
```

CHAPTER 7. EVALUATION

```
4  propertyName = 'Application_idx'
5  for node in output:
6      node['ev'][propertyName] = entityIdx
7      Graph.push(node['ev'])
8      entityIdx += 1
```

<div align="center">Listing 7.4: Create Entity-Specifc Ordering Index (Python)</div>

In line 1 we define the actual Cypher query to get all events with a relationship to one of our *Application* entities and order them by their global ordering index (the original sequence of all events in the log), which is a helper property, like all indexes, to keep track of sequential information and thus can be deleted after use to not affect the schema. The result of this query is saved into the pandas dataframe *output*. Every single event node in the query output becomes a new property *Application_idx* which is a strictly ordered index over the events of the *Application* entity and because the ouput of the query is ordered ("ORDER BY"), we maintain the correct order of events.

Based on the entity-specific index, we can now create the *:DF* relationships for events associated to this entity type.

```
1  MATCH (e1:Event) -[:E_EN]-> (ent:Entity {EntityType: "Application"}) <-[:E_EN]- (e2
       :Event)
2  WHERE e2.Application_idx - e1.Application_idx = 1
3  MERGE (e1) -[df:DF]-> (e2)
4  ON CREATE SET df.EntityTypes = ["Application"]
5  ON MATCH SET df.EntityTypes = CASE WHEN "Application" IN df.EntityTypes THEN df.
       EntityTypes ELSE df.EntityTypes + "Application" END
```

<div align="center">Listing 7.5: Create Entity-Specific :DF Relationships (Cypher)</div>

The listing above matches every pair of events with a relationship to a *Application* entity with *e2.Application_idx - e1.Application_idx* = 1. Since we have a discrete, strictly ordered index, we can derive that *e2* directly follows *e1*. With the *MERGE* clause in line 3 we make sure that for every pair of nodes maximum one *:DF* relationship will be created, regardless for how many entities this relationship it indicates a *:DF*. Through this merge, we ensure that the *:DF* relation adheres to the rules of the *2_df* pattern. Lines 4 and 5 help us to define different actions for different situations in the MERGE clause. As we explained before, the *MERGE* clause creates the relationships only once, but since we also want to maintain a correct list of entities to that relationship, we specify the creation of a new property *df.EntityTypes* with a list with only one entry *["Application"]* if the relationship does not exist, yet ("ON CREATE SET"). If the relationship already existed, we check whether the new entity is already member of the list and if not, it is added ("ON MATCH SET").

The last step in the creation of a general graph event log is creating the *:HOW* relationships of the resources. The information can be derived from two events *e1* and *e2*, the *:DF* relationship between them and the resources *r1* and *r2* related to the events respectively. The we continue with the application example in the query below.

```
1  MATCH (r1:Entity {EntityType: "Resource"}) <-[:E_EN]- (e1:Event) -[rel:DF]-> (e2:
       Event) -[:E_EN]-> (r2:Entity {EntityType: "Resource"})
2  WHERE 'Application' IN rel.EntityTypes
3  MERGE (r1)-[how:HOW]->(r2)
4  ON CREATE SET how.EntityTypes = ["Application"]
5  ON MATCH SET how.EntityTypes = CASE WHEN "Application" IN how.EntityTypes THEN how.
       EntityTypes ELSE how.EntityTypes + "Application" END
```

<div align="center">Listing 7.6: Create Entity-Specific :HOW Relationships (Cypher)</div>

Line 1 matches *r1* and *r2* if they are connected via two events. Line 2 reduces the set of sub-graphs to those where the *:DF* refers to an *:Application* entity. Lines 3-5 follow a similar logic like lines 3-5 in listing 7.5, only for *:HOW* relationships between resources instead of *:DF* relationships between events.

With the above steps, the creation of the graph event log base is completed. The script template in appendix A.6.1 can be used to replicate these results. This graph can now be used as baseline for analyses and to create additional data structures for different types of analyses.

A Schema Framework for Graph Event Data   63

7.3.2 Graph Event Data for BPIC 17

As shown in table 7.2, we identified four entity types and their ID attributes, *Application*, *Workflow*, *Offer* and *Resource*. Furthermore, we have two timestamps. The other attributes are not particularly interesting as they do not contribute to the temporal order or the multidimensionality of the events and their entities.

The data schema consists of the *0_core* pattern (to model events, entities and logs), the *1_df* pattern (to model temporal relations) and the *2_how* pattern (to model resource involvement as this information is present in the BPIC 17 data). The schema pattern definition is exactly the definition of the *2_how* pattern in listing 6.2, because we did not use any of the other patterns on hierarchy level 1. The concrete schema definition can be found in listing 7.7.

The schema in property graph representation looks as follows: We put a focus on keeping the
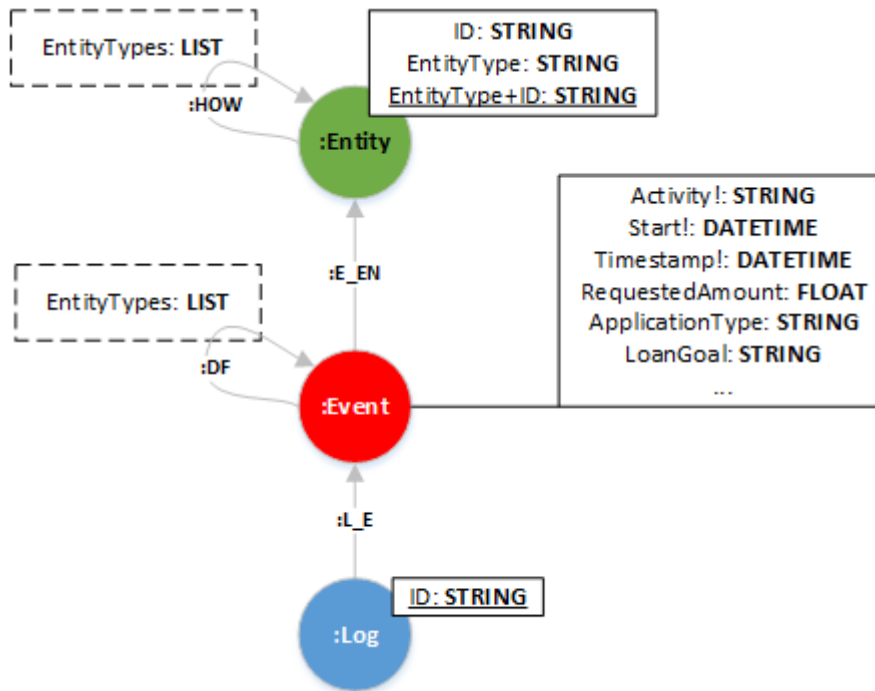
Figure 7.2: BPIC 17 Graph Event Log Schema

structure of the schema as easy and as readable as possible. As reasoned in Chapter 4 and 5, different entity types are *not* represented by different labels, but share the same label *:Entity*; the type information is described as a property of an *:Entity* node. The '...' in the event property box indicates that not all properties are shown in the figure. The schema definition in the listing below gives us the full picture .

```
1  schema_bpic17 = (
2      {//element types
3          Event {
4              Activity!: STRING,
5              Start: TIMESTAMP,
6              End: TIMESTAMP,
7              Timestamp!: TIMESTAMP,
8              Action: STRING,
9              FirstWithdrawalAmount: FLOAT,
10             NumberOfTerms: INTEGER,
11             EventOrigin: STRING,
12             Selected: BOOLEAN,
13             CreditScore: INTEGER,
14             case: STRING,
```

```
15                    LoanGoal: STRING,
16                    resource: STRING,
17                    RequestedAmount: FLOAT,
18                    Accepted: BOOLEAN,
19                    OfferID: STRING,
20                    ApplicationType: STRING,
21                    MonthlyCost: FLOAT,
22                    EventID: STRING,
23                    OfferedAmount: STRING
24                }
25            Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
26            Log {ID}: STRING}
27            E_EN {},
28            L_E {},
29            DF {EntityTypes: LIST},
30            HOW {EntityTypes: LIST}
31        }
32    {//node types
33            (:Event), (:Entity), (:Log)
34        }
35    {//relationship types
36            (:Event)-[:E_EN]->(:Entity),
37            (:Log)-[:L_E]->(:Event),
38            (:Event)-[:DF]->(:Event),
39            (:Entity)-[:HOW]->(:Entity)
40        }
41  )
```

Listing 7.7: BPIC17 Schema Definition

According to listing 7.7, we need to have a *Timestamp* property on every node. Since our data set contains two different timestamps per event, we can actually choose which one we use to satisfy the mandatory property. We did that by duplicating one of the timestamps with a simple query:

```
1  MATCH (e:Event)
2  SET e.Timestamp = e.End
```

Listing 7.8: Set Timestamp Property (Cypher)

As stated in the query, the *end* timestamp is the leading timestamp in the log because the original log is ordered by that timestamp,i.e. *not by the start timestamp*, and the graph event directly follows relationships are generated based on that order. If we want to use a different timestamp for this, we either need to create a dedicated graph instance and predefine the start timestamp as leading timestamp, or we need to define a more complex *:DF* pattern that it can consider different timestamps and allows dedicated *:DF* relationships for the different timestamps similarly to how the current *:DF* pattern treats the different entity types.

7.3.3 Examples for BPIC 17

Additional Data Structures: Create Derived Entity Types

For the BPIC 17 data set, we do not introduce any further structures in terms of patterns such as coinciding events or coinciding entities. What we do, however, is to derive new entity types from the entity types that already exist in the graph. This is to test the entity concept introduced in 2.2.1 on the graph for whether we are able define new entity types from the combinations of other entity types, e.g. to create an entity type for the actual case identifier of the BPIC 17 log from the entity types *Application*, *Workflow* and *Offer*. The task to derive new entity types from existing ones involves the exact same steps like the entity types we created from the imported data. The only difference is that we correlate the event nodes of different entity types or simply select entity types that are candidates for a alternative case identifier in the source log with each other. In this step we defined three new entities: *Case_AWO*, *Case_AO* and *Case_R*. *Case_AWO* is the entity for the actual case identifier from the log. For the case notion of *Case_AWO*, we included *Application*, *Workflow* and *Offer* events, for *Case_AO* only considered event of *Application* and

Offer and for the *Case_R* entity we actually defined our resource entity to be the "case identifier" of the new entity type. All these case definitions coexist in the same graph instance and can be queried separately. In the graph creation section 7.3.1 to this case study we describe the creation of new entities much more detailed.

Say we want to create an entity from the combination of *Applications* and *Offers*, so we first need to define what identifier is suitable for the new entity. In this case, we have a leading entity *Application* since it has a 1:n relation with *Offer* and as stated in table 7.2, we can take the "CaseID" as identifier. We called this new entity type *Case_AO* and created the entity nodes with the following Cypher query:

```
1 MATCH (e:Event) WITH e.CaseID AS id
2 MERGE (:Entity {cbOpen}ID:id, uID:("Case_AO"+toString(id)), EntityType:"Case_AO"{
    cbClose})
```

Listing 7.9: Create Combined Entity Nodes

As you can see, we used the *e.CaseID* property as ID and set the entity's *uID* and *EntityType* properties accordingly. The next step is to relate all those events related to *Offer* or *Application* (but not those related to *Workflow*) to the newly created *Case_AO* entities.

```
1 MATCH (e:Event)−[:E_EN]−>(ent)
2 WHERE (ent.EntityType IN ["Offer","Application"])
3 MATCH (n:Entity {EntityType:"Case_AO"})
4 WHERE n.ID = e.CaseID
5 CREATE (e)−[:E_EN]−>(n)
```

Listing 7.10: Correlate Combined Entity with Events

We first filter in only only those events and their entities that are either *Application* or *Offer*. The second *MATCH* selects the entity nodes of the "combined entity" from step one. This way we can create *:E_EN* relationships between each selected event and the corresponding new *Case_AO* node *n* (by n.ID = e.CaseID).

After the nodes for the combined entity have been created an correlated to the events, we can now follow the standard procedure for creating entity types in the process in figure 7.1, as indicated with the light blue arrow.

This means we can adapt to the new entity type and apply the query in listing 7.4 to create the *Case_AO*-specific order index.
Adapt and apply the query in listing 7.5 to create the *Case_AO*-specific *:DF* relationships and Adapt and apply the query in listing 7.6 to create the *Case_AO*-specific *:HOW* relationships. This all we need to created combined entities. Resources, for example, can also be defined as 'case'. To do so, we only look at the events from the resource perspective, meaning that a case represents all events a resource creates while doing it's tasks, in the correct temporal order. In a graph database, with the structure and schema framework as we introduced it, we can define such cases on the fly. With traditional event log several filter and sort actions are usually required to define a new case.

Figure 7.3 shows the Neo4j graph output on instance level of one complete

We ordered, removed irrelevant (*Resource* entities) and expanded interesting (*Case_AO entity*) elements in the graph view, manually after the query. As you can see, such a simple output can get difficult to lay out very quickly. The entities in the upper part of figure 7.3 are the *Application*, *Workflow* and *Offer* entities that had been created during the log creation process. The lower two entities are *Case_AWO* and *Case_AO*. As can be seen from the *:E_EN* relationships of the two combined entities, *Case_AWO* refers to the events of all three 'basic' entities and *Case_AO* refers only to events of *Application* and *Offer*. The *:DF* relationships of the events are not shown clearly, but as we have five entities for which different *:DF* relationships exist, you can get a rough idea that the number of relationships grows as the number of entities grows. In figure 7.4 we show the same instance we show the same event nodes reordered to illustrate the different *:DF* relationships. Workflow events at the top, application events at the center and offer events at the bottom.
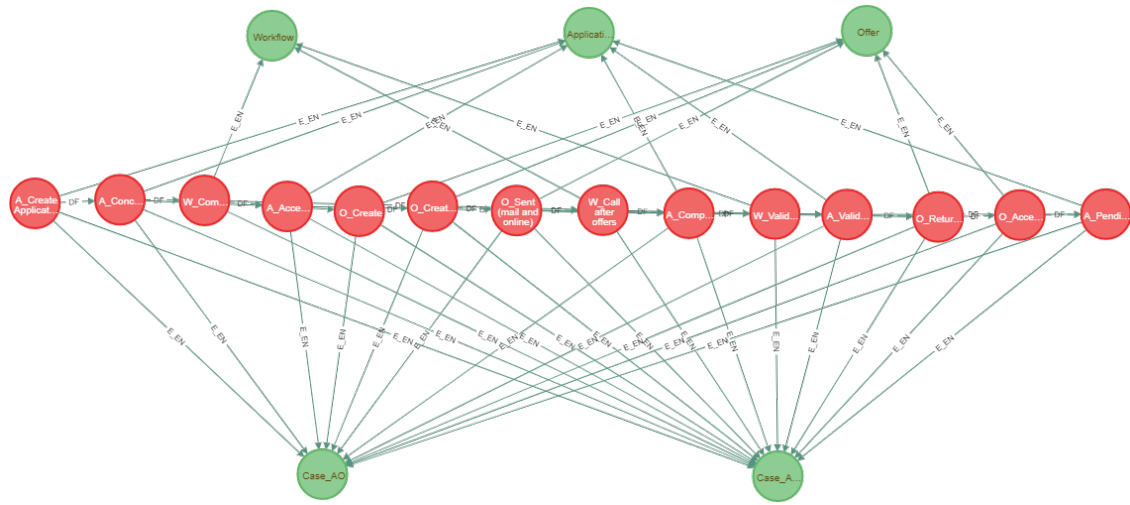
Figure 7.3: Events Related to the 3 Basic Entities and 2 Derived Entities
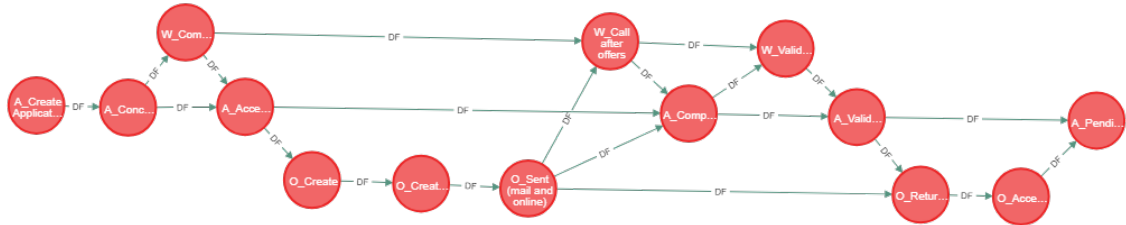
Figure 7.4: Events of Figure 7.3 Reordered

Additional Data Structures: Correlate Coinciding Events

Data convergence is a problem often discussed in process mining literature [20, 28]. As this problem is related to the multi-dimensional attributes of the event data, we want to assess the ability of our solution to enable process mining practices to create correlations between coinciding events. Unfortunately, none of our available data sets contained indicators in the event data whether certain events may coincide or not. Therefore, we decided to reshape the structure of BPIC 17 to artificially create coinciding events and thereby create a second case study based on this data set. Table 7.3 shows a simplified example of the original BPIC 17 data, where each event has two timestamps, one for the start and one for the end of the activity.

In figure 7.5 we show how the corresponding (sub-)graph instance for the original data looks like after creating the graph.

Table 7.4 shows how the log is structured after the new preparation steps and in figure 7.6 we show how the corresponding graph structure changed.

As you can see from the pictures above, we generated longer (doubled) sequences of events

cID	eID	Activity	Start	End	...
1	1	Create Appl.	29.08.19 10:30	29.08.19 10:40	...
1	2	Appl. Ready	29.08.19 10:35	29.08.19 10:41	...
1	3	Appl. Complete	30.08.19 13:59	30.08.19 14:00	...
...	...	...	...	...	...

Table 7.3: BPIC 17 Simplified Example Table With Two Timestamps per Event
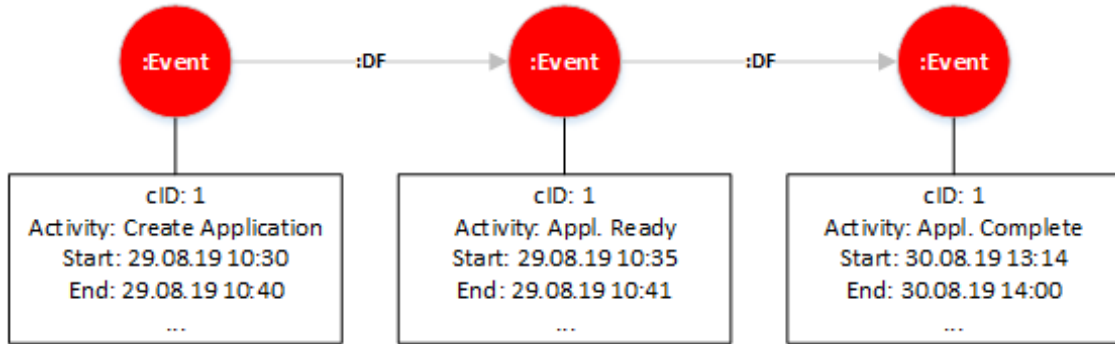
Figure 7.5: BPIC 17 Simplified Example Graph With Two Timestamps per Event

cID	eID	Activity	Timestamp	...
1	1	Create Appl.	29.08.19 10:30	...
1	2	Appl. Ready	29.08.19 10:35	...
1	1	Create Appl.	29.08.19 10:40	...
1	2	Appl. Ready	29.08.19 10:41	...
1	3	Appl. Complete	30.08.19 13:59	...
1	3	Appl. Complete	30.08.19 14:00	...
...	...	...	...	...

Table 7.4: BPIC 17 Simplified Example Table With One Timestamp per Event
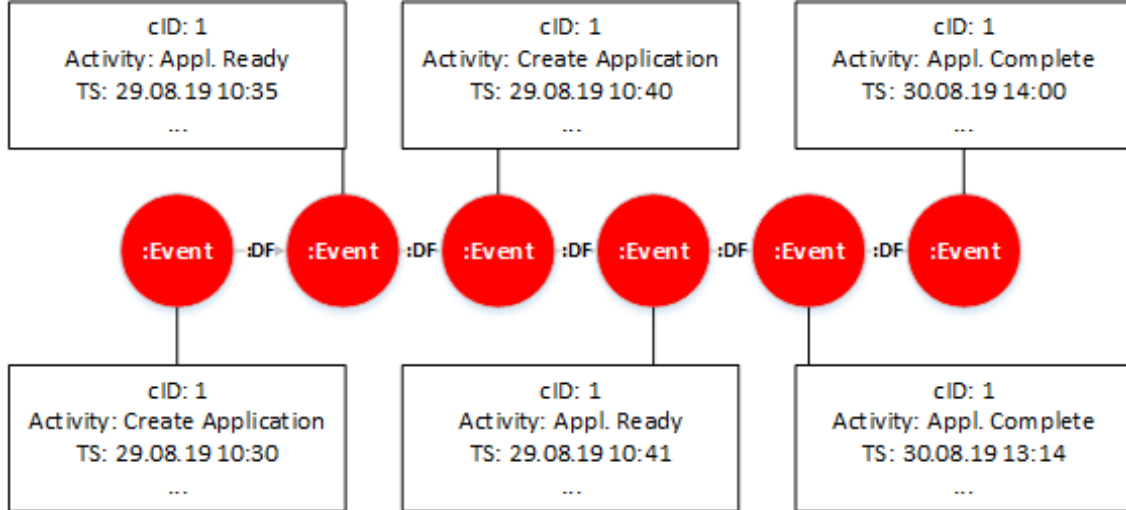
Figure 7.6: BPIC 17 Simplified Example Graph With One Timestamp per Event

through this transformation compared to the original data and we ordered the sequence according to the timestamps. Besides this data preparation step and different "Additional Data Structures", everything regarding the graph event log creation has been done the exact same way as for the original BPIC 17 data. The we added the *1_e_coincide* pattern as described in section 6.3, to be able to correlated the events in the graph accordingly. As for every case study, we followed the process show in figure 7.1 to create the BPIC 17 graph event log with split events. Due to the duplication of the number of events in the source log, the number of event nodes in the graph doubled to 1,123,342 accordingly. The numbers of entity and log nodes did not change, compared

to the first BPIC 17 case study. Creating the basic graph event log took 5.87 hrs and has a database size of 5.71 GB.

Due to the fact that we added a new template, we have to adjust the pattern definition accordingly:

```
pattern_bpic17_split = (
    {//element types
        Event {Activity, Timestamp}
        Entity {ID, EntityType, ID+EntityType},
        Log {ID},
        E_EN {},
        L_E {},
        DF {EntityTypes},
        E_COINCIDE ,
        HOW {EntityTypes}
    }
    {//node types
        (:Event), (:Entity), (:Log)
    }
    {//relationship types
        (:Event)-[:E_EN]->(:Entity),
        (:Log)-[:L_E]->(:Event),
        (:Event)-[:DF]->(:Event),
        (:Event)-[:E_COINCIDE]->(:Event),
        (:Entity)-[:HOW]->(:Entity)
    }
    {//inherited patterns
        0_core, 1_df, 1_e_coincide
    }
)
```

Listing 7.11: BPIC 17 Split Pattern Definition

We marked the newly added elements of the definition with **bold** text in listing 7.11. This of course affects the global schema definition as well:

```
schema_bpic17_split = (
    {//element types
        Event {
            Activity!: STRING,
            Timestamp!: TIMESTAMP,
            Action: STRING,
            FirstWithdrawalAmount: FLOAT,
            NumberOfTerms: INTEGER,
            EventOrigin: STRING,
            Selected: BOOLEAN,
            CreditScore: INTEGER,
            case: STRING,
            LoanGoal: STRING,
            resource: STRING,
            RequestedAmount: FLOAT,
            Accepted: BOOLEAN,
            OfferID: STRING,
            ApplicationType: STRING,
            MonthlyCost: FLOAT,
            EventID: STRING,
            OfferedAmount: STRING
        }
        Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
        Log {ID}: STRING}
        E_EN {},
        L_E {},
        DF {EntityTypes: LIST},
        E_COINCIDE ,
        HOW {EntityTypes: LIST}
    }
    {//node types
```

```
32              (:Event), (:Entity), (:Log)
33          }
34      {//relationship types
35              (:Event)-[:E_EN]->(:Entity),
36              (:Log)-[:L_E]->(:Event),
37              (:Event)-[:DF]->(:Event),
38              (:Event)-[:E_COINCIDE]->(:Event),
39              (:Entity)-[:HOW]->(:Entity)
40      }
41  )
```

Listing 7.12: BPIC17 Split Schema Definition

Event though the changes appear to be relatively small, compared to the original BPIC 17 schema definition, but since the rules for the *1_e_coincide* pattern template now apply to the graph instance, we can now systematically create and query the coinciding event structures.

Based on the *1_e_coincide* pattern, we created collector nodes for each set of coinciding events with the query shown in listing 7.13 and related these nodes based on their event ID ("eID") property accordingly.

```
1  match (e:Event) <-[:L_E]-(l:Log) -[:L_E]->(e2:Event)
2  where e.eID = e2.eID
3  MERGE (collector:Event {ID: e.eID})
4  MERGE (e)-[:E_COINCIDE]->(collector)
5  MERGE (e2)-[:E_COINCIDE]->(collector)
```

Listing 7.13: Query to Correlate Coinciding Events

Note that the *:Log* node used in the pattern is not shown in the example figures 7.5 and 7.6 as they only show sub-graphs of the complete instance. According to the defined schema and the pattern templates used for BPIC 17 in general ensure that this log node must exist when querying the graph event data. For example, for event 1 ("eID") we show the the resulting collector node in figure 7.7.
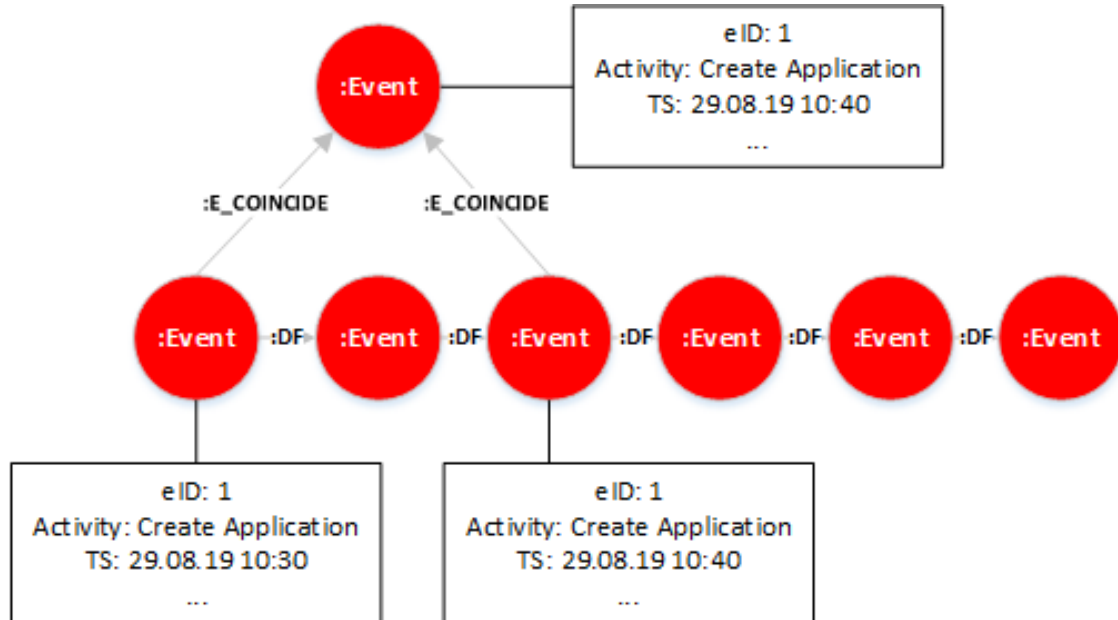

Figure 7.7: BPIC 17 Coinciding Events with Collector Node

If we want to correlate events over different logs, i.e. two events related to different *:Log* nodes, we can adjust the query accordingly.

While the creation of the log with duplicated events has been completed successfully, creating the collector nodes for the complete data set failed. We assume that the graph grew too big for our hardware, as this query would add another 561,671 nodes and 1,123,342 relationships to the graph. We, however, verified the pattern template on a smaller sample of the data with the sample cases encoded in the script template in appendix A.7.1. In principle, the pattern template works as intended, except for the performance.

Querying Behavior Across Multi-Dimensional Instances

In the paper [12], we defined a query over multidimensional data which we also want to test on the structure and the framework defined in this thesis. Say we have an entity type *Case_AWO*, similarly created like *Case_AO* shown above. For that entity type, we want to find all entities that are related to two or more *Offer* entities where event *e1* with e1.Activity = "O_Created" is directly followed, with respect to the *Offer* entity, by *e2* with e2.Activity = "O_Cancelled". For all of these *Offer* nodes, we want to retrieve the path between the very start of the respective *Case_AWO* start event with Activity = "A_Create Application" and every "O_Cancelled" event of offers that match our criteria. This query is hardly doable without plentiful aggregations, sorting and data splitting and recombining on basis of a sequential event log. However, in the paper above we have been able to show that this query is possible without any further tooling other than the native Cypher functionality. The listing below shows the adapted query for our new event log meta model:

```
1 MATCH (o:Entity {EntityType: "Offer"})<-[:E_EN]-(e1:Event {Activity: "O_Created"})
      -[df:DF]-> (e2:Event {Activity: "O_Cancelled"})-[:E_EN]->(o)
2 MATCH (e2)-[:E_EN]->(c:Entity {EntityType: "Case_AWO"})<-[:E_EN]-(e1)-[:E_EN]->(o)
3 WITH c, count(o) AS ct
4 WHERE ct > 1
5 MATCH (o:Entity {EntityType: "Offer"})<-[:E_EN]-(e1:Event {Activity: "O_Created"})
      -[df:DF]-> (e2:Event {Activity: "O_Cancelled"})-[:E_EN]->(o)
6 MATCH (e2)-[:E_EN]->(c)<-[:E_EN]-(e1)-[:E_EN]->(o)
7 RETURN o ,c, e2
```

Listing 7.14: Query Multi-Dimensional Behavior

We have split that query into two parts. The first query in listing 7.14 returns exactly the 218 *Offers* of 103 *Case_AWO* entities where the conditions from the analysis question are met. It also returns the respective *O_Cancelled* events we can then use to query for the paths. The second step, however, poses a challenge to in our new data model as we need to query a path of *:DF* patterns with variable length. To get the respective paths as output, we usually want to use the *-operator with the *:DF* relationship type. Similar to the following listing:

```
1 MATCH p = (A_Created:Event {activity: "A_Create Application"}) -[:DF*]->(e2)
2 RETURN p
```

Listing 7.15: Query Paths for Multi-Dimensional Behavior

The problem is that, with our data model, we have a list of entity types as property of the *:DF* relationships. Filtering on a list in combination with the *-operator was not possible and thus we have not been able to get desired path-based output with the data model specified for this thesis. With the data model used in [12], however, the desired output could be generated.

7.3.4 Schema Validation

With the graph instance in place, we want to validate three things:

1. Does the graph instance conform to the global schema (figure 7.2)?

2. Does the global schema conform to the local, pattern based schema?

3. Does the graph instance conform to the pattern rules?

First, we want to verify that the graph instance and the global schema we defined for this data set shown in figure 7.2 conform. By executing the "db.schema()" function of Neo4j at our graph instance, we receive a proprietary schema like model comparable to our global schema definition (shown in figure 3.1). We show the the output of this function in figure 7.8.
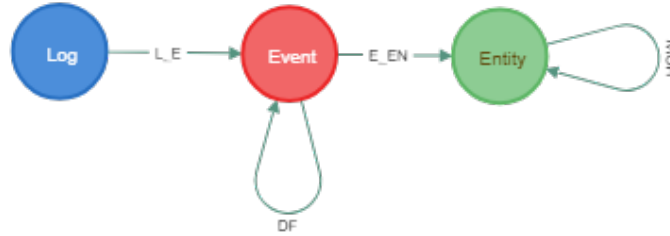

Figure 7.8: Neo4j Schema for BPIC 17

Compared to the defined global schema in figure 7.2, we can at least say that the node and relationship types match. The Neo4j schema, however, does not provide a list of property types per element type in the graph. Thus, we can use the "exists" functions with all element types specified in our global schema to check whether they occur in the database or not. For example, the listing below queries counts the number of events that have an *Activity* property.

```
1 MATCH (e:Event)
2 WHERE exists(e.Activity)
3 RETURN count(e)
```

Listing 7.16: Query Property Exists

The expected output is equal to the number of events in the log, because *Activity* is defined as mandatory property to event nodes. Relationship properties can be queried accordingly. Since we did not have a systematical approach available to check for full compliance, we tested the mandatory (including unique) properties specified in the global schema. With the query

```
1 MATCH (e:Event)
2 RETURN keys(e)
```

Listing 7.17: Query Property Keys of Nodes

we get all property keys of all nodes in the graph. By comparing the distinct values of the output, with the property types in the schema definition in listing 7.7 we can validate that the instance has the same properties we defined. When replacing the "keys(e)" by the "properties(e)" function, we can furthermore validate the datatypes of the different keys. With these manual test we successfully validate that the BPIC 17 graph instance conforms to the global schema.

Second, we want to validate that the global schema conforms to the local schema patterns specified in listing 6.5, as it is equal to the BPIC 17 local pattern schema. To do so, we can take the validated schema definition in listing 7.7 and check if all element types and respective property keys, node types and relationship types in of the local schema (listing 6.5) are also included in the global schema (listing 7.7). As this is the case, we validated the conformance of the global schema to the local pattern schema.

Third, we want to verify that the graph instance conforms to the pattern rules. The verification of the rule compliance is very specific to the type of rule and the complexity of the individual rules. For every rule, we can define 'test cases' with expected outcomes and define respective queries that should produce the expected outcomes, otherwise we institute a non-compliant data model. For example, we want to verify that rule 2 of the core pattern in section 6.1 is not violated. The rule specifies that and event node e can have maximum one incoming *:L_E* relationship. Therefore, we define a query that identifies all event nodes with more than one *:L_E* relationship and expect the query to return an empty set. We query:

```
1 MATCH () −[:L_E]−>(e:Event)
2 WITH count(e) as eventCount, e
```

Entity Type	# of Event Nodes	# of Entity Nodes
All	561,671	169,312
Application	239,595	31,509
Workflow	128,227	31,500
Offer	193,849	42,995
Case_AWO	561,671	31,509
Case_AO	433.444	31,509
Resource	561,671	145
Case_R	561,671	145

Table 7.5: BPIC 17 Event Log Entities

```
3  WHERE eventCount > 1
4  RETURN eventCount , e
```
Listing 7.18: Example Query for Violated Cardinalities Rule

The result for BPIC 17 was empty, thus we imply that the rule is respected.

Another example of a rule validation is rule 1 of the directly follows pattern in section 6.2. It specifies that a *:DF* relationship may only exist between two *:Event* nodes *e*1 and *e*2 with $e1 \neq e2$. The query below returns all source and target nodes *n* of *:DF* relationships with a label other than *:Event*. Empty output is expected for compliance.

```
1  MATCH (n) −[:DF]−(:Event )
2  WHERE NOT ”Event” IN labels (n)
3  RETURN n
```
Listing 7.19: Example Query for Violated DF Rule

The BPIC 17 graph event log passed this and all further tests we ran against it. Therefore, we claim the graph event log is compliant to the entire graph schema (global schema, local patterns and rules).

This validation, however, was not conducted systematically and we do not claim to have covered 100% of possible violations, but we tried to be as complete as possible in the given scope of this thesis. A systematic schema validation for our schema framework might be subject to future research.

7.3.5 Graph Event Data Statistics

In table 7.5 we summarize statistics of the different entities. These have been retrieved by querying the graph event data and counting the number of events and entities. The number of entity and event nodes are feasible. The entity and event numbers of the base entities are equal to the ones of the original log as stated earlier in this section. The sum of entity nodes over all entity types sums up to the number of all entities, which means that there are no entities created unintentionally. The sum of *Application*, *Workflow* and *Offer* events is equal to the number of the query result for all event nodes in the graph.

Creating the graph took 3.94 hours on our test system. The Neo4j database has a size of 6.34 GB with all elements described above. The original CSV log file (before the data preparation) has a size of 0.28 GB which is loaded into process mining tool such as ProM in a matter of seconds.

7.4 Other Data Sets

In this section, we want to briefly report on the findings of the four other case studies we conducted in the course of this thesis. Generally, all case studies had the same setup as described in section 7.1 and had their execution was very similar to the execution of the BPIC 17 case study described in

Log	# of Cases	# of Events	# of Entity Types
Change	?	30,275	3
Incident	?	46,606	6
IncidentDetail	46,616	466,737	5
Interaction	?	147,004	5

Table 7.6: BPIC 14 Event Log Events & Entities

detail in section 7.2. To avoid too much repetition, we mainly elaborate on interesting observations in terms of differences and similarities to the case studies described above.

7.4.1   ITIL Service Management Process

The BPIC 14 [29] data set contains four logs of an ITIL service management process. One log with change related events, a log for incident events, another log containing details to the incidents and the fourth log contains events regarding user interactions. We have been able to create a graph event log for the full data set. For performance reasons, we had to limit the number of entities encoded in the graph. The result is a graph event log with 1.155.019 nodes and 3.140.289 relationships. Its database has a size of 3.94 GB and the creation script template shown in appendix A.3.1 took 44.04 hrs to complete. In table 7.7 we show the event and entity statistics of the log.

The number cases for the all logs, but the *IncidentDetail* log is marked with a question mark. This is due to the fact that we don't have a clear activity notion in these logs. For the incident details, there is an activity notion in the log. Therefore, we treated the incident details as the 'leading' log here as it is the only log that fits our definition of an event log. For the other logs, we would need to randomly choose an activity notion to comply with the log's graph schema shown in appendix A.4.2. This raises the question whether we need a fixed activity notion for graph event data or not, because we have been able to correlate events to entities and we can specify any property of the event nodes as activity at query time. Whats interesting about this data set is that it contains different log files. This means, that our global schema grows in terms of event properties. Logs that have dedicated event attributes add properties to the global schema that events of other logs do not have, for example the activity property as mentioned above.

Besides the activity, the log is fairly interesting from a multi-dimensional process perspective. Different logs have different entity types in common which is a good use case for our pattern template to correlate coinciding entities. After creating the base log, we used queries like shown in listing 7.20 to create the collector node for entities across and within logs.

```
1 match (en:Entity) where en.Log = 'CHG'
2 match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
      and en2.Log = 'INC'
3 merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
      .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
4 merge (en)-[:EN_COINCIDE]->(collector)
5 merge (en2)-[:EN_COINCIDE]->(collector)
```
Listing 7.20: Query to Correlate Coinciding Entities

This query correlates the entities of the change log ("CHG") with the entities of the incident log ("INC"). For every pair of logs we must run such a query to capture all coinciding events across logs.

7.4.2   Building Permit Application Processes

BPIC 15 [30] contains data of the same type of process of five municipalities, but with slight variations for each municipality. These five processes event logs have a rather flat structure and because the different municipalities don't interact during these processes, there are no relation

Log	# of Cases	# of Events	# of Entity Types
Log 1	1,199	52,217	6
Log 2	832	44,354	6
Log 3	1,409	59,681	6
Log 4	1,053	47,293	6
Log 5	1,156	59,083	6

Table 7.7: BPIC 15 Event Log Events & Entities

Log	CSV Log Size
Click logged in	1.11 GB
Click not logged in	1.14 GB
Messages	<0.01 GB
Questions	0.03 GB
Complaints	<0.01 GB

Table 7.8: BPIC 16 Event Log Events & Entities

between entities or events of the different logs. Table 7.7 gives an overview of the entities and events of the different logs.

With the log creation script in appendix A.4.1, we created the graph event log in 0.56 hours with a size of 1.68 GB. The graph event log consists of 268,354 nodes and 1,032,155 relationships.

The resulting global and local schemata can be found in appendix A.4.2.

This process shows rather standard behaviour in terms of multi-dimensional data.

7.4.3 Customer Communication Process

BPIC 16 [31] consists of five different event logs, all containing a sub-process of a customer communication process. One log containing click data on a website of users not logged in. A second log containing click data for users logged in. The other three logs contain event data about messages, questions and complaints of customers. This data set is bigger than any of the other data sets. To reduce the amount of data, we dropped the data set with users that are not logged in. This is because we assumed that without being able to correlate these clicks to any of the other data, the unrelated data will not be interesting for us. Table 7.8 gives an overview over the size of the data.

Even though we spared out one of the clicks logs already, the creation of the graph event log from the full data set was still not possible. Without the two clicks logs, we created the graph with the script shown in appendix A.5.1. Creating the graph took 9.37 hrs and the database is 2.53 GB big.

The global and local schema can be found in appendix A.5.2.

7.4.4 Purchase Order Handling Process

BPIC 19 [33] contains the data of a purchase order process from a SAP data source. The main entities of the process are purchase orders (PO) and purchase order items (POI). From the dimensionality perspective, the process structure is very similar to BPIC 17 as PO and POI have a 1:n relationship, just as application and offer in BPIC 17. With a size of 1,595,923 events, this data set seems to close to infeasibility in terms of performance. The graph, created with the script in appendix A.8.1, has 1,848,914 nodes and 7,737,347 relationships. We assume that selecting POI as entity lead to the high number of relationships, as there are many entities with only a few events each. The graph was has been created with a waiting time of 82.90 hrs and reached a size of 14.90 GB.

The global and local schema are shown in appendix A.8.2.

7.5   Summary

To summarize the findings of our case studies we first conclude in our evaluation objectives:

1. We have been able to construct graphs with the schema and pattern definitions in Chapters 3 and Chapter 4 for all event logs we studied.

2. All schema patterns proposed in Chapter 5 could be applied in the case studies. For analyses and queries, however, some limitations noticed that require adaption of the queries.

3. The entity concept has been successfully implemented by creating new case notions from existing base patterns.

4. We have been able to query the data in a meaningful manner, however, not all queries could be completed successfully.

5. The performance of the graph event data compared to its CSV or XES counterparts is significantly worse. The processing time and the data size is much higher.

The limitations in querying have been caused by our pattern definition. The issue with the list property and the *-operator can for example be solved by defining the *:DF* pattern in a way that for every entity type, a dedicated *:DF* relationship is created between two events *e1* and *e2*. This will lead to more (relationship) objects in the database, but we expect for medium sized event logs such as BPIC 17 the additional load can still be handled by a standard laptop.

Interestingly, the visual schema representations in our case studies all look very similar. This is due to the very generic way we defined the patterns, i.e. there exists only one node type for all entities. Compared to the data model in [11] shown in figure 4.2, many objects have been merged and use the same element types in our new data model. Furthermore, each pattern, except for *0_core*, only adds a very small portion of actual graph elements, but on the other hand enriches the model with rules that do not have any impact on the visual pattern or schema representation.

The *0_core* pattern and the *1_df* pattern have been used in every case study. We may consider the integration of the *1_df* pattern into the core *0_core* in future developments, because the temporal order of events is crucial for event data and thus will probably always be needed.

Chapter 8

Conclusions

This chapter concludes the thesis. First, we summarize and discuss our results and findings in section 8.1. The limitations of our work and possible future developments based on the outcomes are discussed in section 8.2

8.1 Results & Discussion

In this thesis, we explained the need for an event data model that supports multi-dimensional relations of events and entities. Our proposed solution for this problem, property graphs, proofed suitable in preliminary research [11, 12] to this thesis. However, this work lacked a formalized schema for graph event data and thus does not provide a sufficient basis for modelling and sophisticated analyses of various event data sets. The lack of a schema standard for property graphs and the need to derive case identifiers beyond the attributes you just see in the data, e.g., by combining two attributes, constituted additional challenges for our research.

We followed the design science research process to develop a set of artifacts to help us in achieving the research goals set in section 1.2 in accordance with the requirements specified in section 1.3.

Our first contribution is a global graph schema definition to describe the data structure of entire property graph instances in Chapter 3 based on the work of Bonifati et al. [5]. The corresponding design artifact consists of a schema definition language and a visual representation of the schema. With this schema definition approach we are able to describe all node types, relationship types and properties with their data types of a graph database instance. Furthermore, we are able to define unique constraints to property types and to define mandatory property types. As we are not capable of defining behavioral attributes, i.e. fully encode event log concepts, in the schema, we partly fulfill research goal G1.

The second contribution of this thesis is a non-exhaustive catalogue of event log concepts encoded in the property graph data model described in Chapter 4. We defined how events or directly follows relations can be transformed into property graph concepts and thus fulfilled G2.

Our third contribution is actually a complement artifact, or rather a set of artifacts to the global schema definition. It was necessary, because the global schema could not encode integrity constraints to the graph schema. In Chapter 5 we introduce local, pattern based schema definitions and a set corresponding schema templates for event data in Chapter 6. Such templates consisting of schema patterns and sets of formal rules and constraints can be used to define mandatory element types to the global schema. They are also used to define rules and constraints to a corresponding graph instance such that the local schema pattern templates with their rules. By combining the local schema pattern templates with the global schema definition, we meet G1 as we now can define a full schema with constraints and rules. Furthermore, with the contributions discussed so far, we meet the requirements R1 and R2 for our design artifacts, as we now can describe a data model of a LPG with event data (R1) and can encode behavioral attributes to

enforce concepts such as handover of work and directly follows (R2).

This Thesis' fourth contribution is a basic method to validate the conformance demands in our schema framework as illustrated in figure 5.5. We provide a non-formalized collection of steps that can be performed to validate the conformance of a graph instance to a schema by successively querying the instance for all element types in the schema. We realized the conformance checking of the global schema towards the pattern templates rather intuitively, i.e. by visually comparison of the designed templates and the schema representation of the graph database in Neo4j. The conformance of graph instances to the template rules has been validated by custom queries to check for existence or non-existence of structures in the data model that have been prohibited or requested by some rule. Also these checks have not been conducted systematically, but as accurate as possible for the given project schedule, i.e. the combination of validation queries and visual inspections of (sub-)graph instances are deemed sufficient, but without a final proof. Therefore we declare as G3 fulfilled with limitations.

The fifth and last contribution of this thesis is the introduction of a flexible, generic entity concept which allows us to specify single entities, combinations of entities and also complementary entities, such as resources, as case notions. This enables us to encode various relations between entities and events. The entity concept can also be used to encode meta entities, e.g. resources, which provide meta information to events, but usually are not part of a process workflow directly. With the introduction of the entity concept, we meet our research goal G4. This artifact satisfies R4 as the entity concept allows to establish multiple case notions in a single log. We had to realize though, that keeping a single notion of activity raises new questions in terms of multi-dimensional data. Especially in event data sets that consist of multiple event logs, a single activity notion, i.e. every event log in the data set has the same activities, limits the possible views on the data. We realized, that we can actually ignore the activity notion of the source log and relate events to the respective entities at graph creation time, leaving the choice for the activity notion open until the graph event data is queried.

For the sixth type of artifact, we applied all above contributions to develop templates with Python and Cypher for each of the case studies to automatically transform and import the CSV logs to a graph database, i.e. create a graph event log. Actually, these scripts were the main tool for developing the above graph and schema concepts for event logs in more than 100 iterations. For each case study described in this thesis, we provide a a dedicated Python script that can be used to generate a graph event log from the respective CSV log. Please refer to appendix A for a detailed description.

As seventh and last contribution we have implemented graph event logs for all case studies by using the script templates. These graph event logs have furthermore been used the evaluation steps in Chapter 7. For some data sets, we have not been able to create the graph event logs from the full data. This was due to performance issues, i.e. if the time to run the respective Python template took more than 3 days, the execution was cancelled. Such Performance issues have not only been recognized in relation to the number of events in a log file, but also in relation to the number of entity types we wanted to include in the log. On the other hand, when the graph event log has been created successfully, querying these data could still be done with short waiting times. Therefore, the performance of graph event data - at least at creation time - was not competitive compared to classical event logs. We will discuss more on that matter in section 8.2.

During our evaluation on 5 different data sets and throughout the iterations of the case studies, we have been able to optimize the different components of our approach in such a way that we have been able to meet all research goals. We furthermore have shown, with process mining queries in Chapter 7 that we can also fulfill R3 to our design artifacts. In general, the interplay of the different components of our framework was very good and extension of the templates or their adoption to different analysis requirements is expected to work fluently. We, however, had to realize that some of the data structures in the templates have not been designed optimally.

8.2   Limitations & Future Work

First and foremost we want to discuss the performance of the graph event logs. With performance, we mean the database size and the execution time for the graph creation. Even though, once created, the graph database can be queried very quickly, the creation process could not be finished in all case studies. In some of the studies we realized that some of the event logs are simply too big to be entirely loaded into a Neo4j graph database with the data models proposed by us. For one event log (the ITIL process in BPIC 14) we have been able to optimize the loading time by reducing the number of entities to be included in the creation process. For another data set (BPIC 16), we had to remove a specific log file from the load (a log of click data of website users). Even for the data sets that have entirely been imported, the performance in terms of used disk space and loading time was a lot higher than with CSV or XES based event logs. However, we expect that the performance can be increased, especially in terms of loading times by optimizing the queries for the graph creation. Optimizing the queries, or graph creation process in general, is a potential goal for further research.

A further limitation of our work is the validation which has been done almost fully manual. This thesis only required a basic form of validation, because we needed to verify whether our assumptions and expectations towards graph instances, schemata, patterns and rules hold. Our approach was rather pragmatic than systematic since we have validated mainly by visual inspections and to the best of our knowledge. Formally writing the rules and constraints down, however, was necessary to enable a validation in the first place. The rules also allowed us to define individual queries to inspect the graph instances for desired, or undesired data structures. The development of a full validation methodology for our framework was outside the scope of this thesis. Therefore, a potential next step towards a fully functional schema framework for property graphs is to develop a systematic validation approach for our framework.

The next limitation is related to how we used our framework. In essence, we used the tool set of our framework, i.e. the global schema definition and the local pattern templates with their rules, as intended. What we realized during our case studies, however, is that our schema pattern definitions included some design decisions that were not optimal for some more advanced use cases. For example, the *:DF* pattern, which we defined as a relationship between two event nodes *e1* and *e2* with a list property. This list property shall contain all entity types that the *:DF* relationship is valid for. Listing 7.14 and the subsequent explanation shows a good example on how this conception limits our ability to query the event data as required. The positive thing about this limitation is that our framework follows a modular approach and allows to define new or changed local pattern templates. By changing the *:DF* relationship definition in a way that the pattern allows (or enforces) the creation of a distinct *:DF* relationship for every entity type, with a string property only for one entity type. Such that from a relationship

```
(e1:Event)−[:DF {EntityTypes: ["Application","Offer"]}]−>(e2:Event)
```

we create two separate relationships:

```
(e1:Event)−[:DF {EntityType: "Application"}]−>(e2:Event)
(e1:Event)−[:DF {EntityType: "Offer"}]−>(e2:Event)
```

This, however, would further increase the size of such a database, but would enable us to run the query in listing 7.14 and extend it to return the full paths, which is what we intended initially.

The last limitation we want to point out is that, similar to our previous work [12, 11] on graph event data, the findings and concepts from this thesis are based on event log data sets that have been created and thus also flattened already. Meaning that the log extraction part from the actual source of events, i.e. the information system, cannot be put into the context of our framework yet, as a transformation of such "native" event data from another source, such as a relational database, has not been evaluated yet. Such a direct extraction from a RDB and transformation to a PGDB would require an entirely different graph log creation process. However, we assume that our framework may still be applicable to this kind of data as well. We assume that further pattern templates, such as modeling the relationships between different entity types directly, might become handy in such a use case. Our view on the data was event centric and thus there was

no need for such a template, but in RDBs, entities are the predominant logical unit. A direct graph log extraction and a subsequent investigation whether the framework can be used with it as well are two interesting steps ahead. Within such a scope, it might also be interesting to assess whether defining a more flexible activity concepts, e.g. assign an activity notion to an entity type on query time, would be a valuable extension.

Furthermore, we would like to encourage researchers from other fields to explore the use of our framework. The framework provides a tool set that can be flexibly adapted to other domains and different types of data.

Bibliography

[1] Lance Ashdown and Tom Kyte. Oracle database concepts, 11g release 2 (11.2) e16508-04. 16

[2] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986. 16

[3] Alessandro Berti, Sebastiaan J van Zelst, and Wil van der Aalst. Process mining for python (pm4py): Bridging the gap between process-and data science. *arXiv preprint arXiv:1905.06169*, 2019. 57

[4] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. Querying graphs. *Synthesis Lectures on Data Management*, 10:1–184, 10 2018. 2, 3, 10

[5] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. Schema validation and evolution for graph databases. *arXiv preprint arXiv:1902.06427*, 2019. 2, 3, 10, 17, 19, 30, 77

[6] Alessio Bottrighi, Luca Canensi, Giorgio Leonardi, Stefania Montani, and Paolo Terenziani. Trace retrieval for business process operational support. *Expert Systems with Applications*, 55:212–221, 2016. 1

[7] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009. 2, 8

[8] E Gonzalez Lopez de Murillas. Process mining on databases: extracting event data from real-life data sources. 2019. 2

[9] E González López De Murillas, Hajo A Reijers, and Wil MP Van Der Aalst. Connecting databases with process mining: a meta model and toolset. In *Enterprise, Business-Process and Information Systems Modeling*, pages 231–249. Springer, 2016. 2

[10] Daniel Deutch and Tova Milo. Top-k projection queries for probabilistic business processes. In *Proceedings of the 12th International Conference on Database Theory*, pages 239–251, 2009. 1

[11] Stefan Esser. Using graph data structures for event logs. Capita selecta research project., Eindhoven University of Technology, 2019. `https://doi.org/10.5281/zenodo.3333831`. 2, 3, 4, 55, 60, 76, 77, 79

[12] Stefan Esser and Dirk Fahland. Storing and querying multi-dimensional process event logs using graph databases. In *International Conference on Business Process Management*, pages 632–644. Springer, 2019. 2, 3, 4, 9, 10, 22, 23, 55, 60, 71, 77, 79, 86

[13] Dirk Fahland. Describing behavior of processes with many-to-many interactions. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 3–24. Springer, 2019. 2, 9

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018. 7, 11, 13, 31

[15] Shirley Gregor and Alan R Hevner. Positioning and presenting design science research for maximum impact. *MIS quarterly*, pages 337–355, 2013. 4

[16] Christian W Günther and Eric Verbeek. Xes standard definition. *Fluxicon Process Laboratories (November 2009)*, 2014. 8

[17] Mieke Jans and Pnina Soffer. From relational database to event log: Decisions with quality impact. In *BPM 2017 Workshops*, volume 308 of *LNBIP*, pages 588–599. Springer, 2017. 2, 8

[18] Mahesh Lal. *Neo4j graph data modeling*. Packt Publishing Ltd, 2015. 10

[19] Dong Liu, Carlos Pedrinaci, and John Domingue. Semantic enabled complex event language for business process monitoring. In *Proceedings of the 4th International Workshop on Semantic Business Process Management*, pages 31–34, 2009. 1

[20] Xixi Lu, Marijn Nagelkerke, Dennis van de Wiel, and Dirk Fahland. Discovering interacting artifacts from erp systems. *IEEE Transactions on Services Computing*, 8(6):861–873, 2015. 2, 8, 9, 60, 67

[21] Anil Nigam and Nathan S Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003. 2, 8

[22] Ken Peffers, Tuure Tuunanen, Charles E Gengler, Matti Rossi, Wendy Hui, Ville Virtanen, and Johanna Bragge. The design science research process: Amodel for producing and presenting information systems research. In *First International Conference on Design Science Research in Information Systems and Technology*, pages 83–16, 2006. 4

[23] Viara Popova, Dirk Fahland, and Marlon Dumas. Artifact lifecycle discovery. *International Journal of Cooperative Information Systems*, 24(01):1550001, 2015. 2, 8, 9

[24] Margus Räim, Claudio Di Ciccio, Fabrizio Maria Maggi, Massimo Mecella, and Jan Mendling. Log-based understanding of business processes through temporal logic query checking. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 75–92. Springer, 2014. 1

[25] Liang Song, Jianmin Wang, Lijie Wen, Wenxing Wang, Shijie Tan, and Hui Kong. Querying process models based on the temporal relations between tasks. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*, pages 213–222. IEEE, 2011. 1

[26] Yan Tang, Isaac Mackey, and Jianwen Su. Querying workflow logs. *Information*, 9(2):25, 2018. 1

[27] Wil MP van der Aalst. *Process mining: data science in action*. Springer, 2016. 1, 7, 8, 35, 49

[28] Wil MP van der Aalst. Object-centric process mining: Dealing with divergence and convergence in event data. In *International Conference on Software Engineering and Formal Methods*, pages 3–25. Springer, 2019. 2, 8, 9, 67

[29] B.F. van Dongen. BPI Challenge 2014. Dataset. `https://doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35`. 74, 85

[30] B.F. van Dongen. BPI Challenge 2015. Dataset. `https://doi.org/10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1`. 74, 85

[31] B.F. van Dongen. BPI Challenge 2016. Dataset. `https://doi.org/10.4121/uuid:` `360795c8-1dd6-4a5b-a443-185001076eab`. 75, 85

[32] B.F. van Dongen. BPI Challenge 2017. Dataset. `https://doi.org/10.4121/uuid:` `5f3067df-f10b-45da-b98b-86ae4c7a310b`. 2, 21, 58, 85

[33] B.F. van Dongen. BPI Challenge 2019. Dataset. `https://doi.org/10.4121/uuid:` `d06aff4b-79f0-45e6-8ec8-e19730c248f1`. 75, 85

[34] Carlo Zaniolo and Miachel A Meklanoff. On the design of relational database schemata. *ACM Transactions on Database Systems (TODS)*, 6(1):1–47, 1981. 16

Appendix A

Case Studies

This appendix provides detailed information about the implementation of our case studies. The below Python scripts have been used to create the graphs for the different case studies. The following sections describe in detail how we conducted the case studies.

A.1   How to

The case studies have been conducted using the following tools:

- Neo4j Desktop 1.2.1 with database version 3.5.9

- ProM Lite 1.2

- Python 3.7.4

    - pandas 0.24.2
    - py2neo 4.3.0

For reproduction of the case studies, perform the following steps:

1. Get one of the BPI Challenge data sets [29, 30, 31, 32, 33].

2. (If applicable) Use ProM Lite 1.2 to convert the XES file to CSV format.

3. Put the corresponding Python scripts from below on your local system.

4. Place the CSV log(s) in the same folder as the Python script.

5. In Neo4j Desktop, create a new database 3.5.9 or higher.

6. In the Neo4j DB settings, change the "dbms.directories.import" parameter of the database to "c:/temp/import".

7.

8. Create a folder "c:/temp/import", or change the path in the script respectively.

9. Run the Python script.

Use a dedicated Neo4j database for each script. All data of a target database will be deleted with the script execution. Configure each script by setting the $sample = True$ for the fixed sample cases and $sample = False$ to load the entire log files.

A.2 XES to CSV conversion

The BPIC17 log conversion with ProM Lite 1.2 from XES to CSV format lead to some unexpected data in the CSV log. We had to realize that different tools, e.g. ProM or Disco, may convert XES files differently. Please refer to the example screenshots below that show differences how certain attributes are converted between the formats and for a number of events the results have not been as we expected. The same issue has been described in the precursor of this thesis [12]. Interestingly, this issue seems to be related to the structure of the event data in the source format, because the data conversion issues occurred on the offer entity and its attributes. Furthermore, BPIC17 is the only log where we had this issue, so it only had to be considered in the data preparation for the two case studies on this data set. Figure A.1 shows an example case of the original log in XES format loaded with Disco. Figure A.2 contains the same case after the conversion to CSV. In the CSV log we find duplicates of the attributes that, logically, belong to the entity Offer in of Workflow or Application related rows (events). This issue has been considered during the data preparation for the case studies on BPIC 17.

Figure A.1: BPIC17 Loaded from XES

Figure A.2: BPIC17 Loaded from CSV

A.3 BPIC14

A.3.1 Script Template

A.3.2 Schema Definitions

```
1  pattern_bpic14 = (
2      {//element types
3          Event {Activity, Timestamp}
4          Entity {ID, EntityType, ID+EntityType},
5          Log {ID},
6          E_EN {},
7          L_E {},
```

```
8             DF {EntityTypes},
9             EN_COINCIDE {}
10      }
11      {//node types
12            (:Event), (:Entity), (:Log)
13      }
14      {//relationship types
15            (:Event)-[:E_EN]->(:Entity),
16            (:Log)-[:L_E]->(:Event),
17            (:Event)-[:DF]->(:Event),
18            (:Event)-[:EN_COINCIDE]->(:Event)
19      }
20      {//inherited patterns
21            0_core, 1_df, 1_en_coincide
22      }
23 )
```

Listing A.1: BPIC 14 Pattern Definition

```
1 schema_bpic14 = (
2      {//element types
3            Event {
4                  Activity!: STRING,
5                  Timestamp!: TIMESTAMP,
6                  start: TIMESTAMP,
7                  end: TIMESTAMP,
8                  IncidentID: STRING,
9                  IncidentActivityType: STRING,
10                 AssignmentGroup: STRING,
11                 KMNo: STRING,
12                 InteractionID: STRING,
13                 ServiceComponentAff: STRING,
14                 CINameAff: STRING,
15                 CITypeAff: STRING,
16                 CISubTypeAff: STRING,
17                 ClosureCode: STRING,
18                 FirstCallResolution: STRING,
19                 HandleTime: INTEGER,
20                 Impact: INTEGER,
21                 Urgency: INTEGER,
22                 Priority: INTEGER,
23                 FirstCallResolution: STRING,
24                 RelatedIncident: STRING,
25                 AlertStatus: STRING,
26                 NoReassignments: INTEGER,
27                 ReopenTime: TIMESTAMP,
28                 ResolvedTime: TIMESTAMP,
29                 NoRelatedInteractions: INTEGER,
30                 RelatedInteraction: STRING,
31                 NoRelatedIncidents: INTEGER,
32                 NoRelatedChanges: INTEGER,
33                 CINameCBy: STRING,
34                 CITypeCBy: STRING,
35                 CISubTypeCBy: STRING,
36                 ServiceComponentCBy: STRING,
37                 ChangeID: STRING,
38                 ChangeType: STRING,
39                 RiskAssessment: STRING,
40                 EmergencyChange: STRING,
41                 CABApprovalNeeded: STRING,
42                 PlannedStart: TIMESTAMP,
43                 PlannedEnd: TIMESTAMP,
44                 ScheduledDowntimeStart: TIMESTAMP,
45                 ScheduledDowntimeEnd: TIMESTAMP,
46                 ActualSTart: TIMESTAMP,
47                 ActualEnd: TIMESTAMP,
48                 RequestedEndDate: TIMESTAMP,
49                 OriginatedFrom: STRING
```

```
50              }
51              Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
52              Log {ID}: STRING}
53              E_EN {},
54              L_E {},
55              DF {EntityTypes: LIST},
56              EN_COINCIDE {}
57          }
58      {//node types
59              (:Event), (:Entity), (:Log)
60          }
61      {//relationship types
62              (:Event)-[:E_EN]->(:Entity),
63              (:Log)-[:L_E]->(:Event),
64              (:Event)-[:DF]->(:Event),
65              (:Event)-[:EN_COINCIDE]->(:Event)
66          }
67  )
```

Listing A.2: BPIC 14 Schema Definition

A.4   BPIC15

A.4.1   Script Template

A.4.2   Schema Definitions

```
1  pattern_bpic15 = (
2      {//element types
3              Event {Activity, Timestamp}
4              Entity {ID, EntityType, ID+EntityType},
5              Log {ID},
6              E_EN {},
7              L_E {},
8              DF {EntityTypes},
9              HOW {EntityTypes}
10         }
11     {//node types
12             (:Event), (:Entity), (:Log)
13         }
14     {//relationship types
15             (:Event)-[:E_EN]->(:Entity),
16             (:Log)-[:L_E]->(:Event),
17             (:Event)-[:DF]->(:Event),
18             (:Entity)-[:HOW]->(:Entity)
19         }
20     {//inherited patterns
21             0_core, 1_df, 2_how
22         }
23 )
```

Listing A.3: BPIC 15 Pattern Definition

```
1  schema_bpic15 = (
2      {//element types
3              Event {
4                  Activity!: STRING,
5                  Start: TIMESTAMP,
6                  End: TIMESTAMP,
7                  Timestamp!: TIMESTAMP,
8                  resource: INT,
9                cID: INT,
10                 event: STRING,
11                 termName: STRING,
12                 startDate: TIMESTAMP,
13                 endDate: TIMESTAMP,
```

```
14              caseProcedure: STRING,
15              Responsible_actor: STRING,
16              caseStatus: STRING,
17              Includes_subCases: STRING,
18              parts: STRING,
19              requestComplete: STRING,
20              last_phase: STRING,
21              landRegisterID: STRING,
22              SUMleges: FLOAT,
23              IDofConceptCase: INT,
24              planned: TIMESTAMP,
25              dateStop: TIMESTAMP,
26              dateFinished: TIMESTAMP,
27              question: STRING,
28              dueDate: TIMESTAMP,
29              monitoringResource: INT,
30
31          }
32        Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
33        Log {ID}: STRING}
34        E_EN {},
35        L_E {},
36        DF {EntityTypes: LIST},
37        HOW {EntityTypes: LIST}
38    }
39    {//node types
40        (:Event), (:Entity), (:Log)
41    }
42    {//relationship types
43        (:Event)-[:E_EN]->(:Entity),
44        (:Log)-[:L_E]->(:Event),
45        (:Event)-[:DF]->(:Event),
46        (:Entity)-[:HOW]->(:Entity)
47    }
48 )
```

Listing A.4: BPIC 15 Schema Definition

A.5   BPIC16

A.5.1   Script Template

```python
1 #website click data, labour services process
2 import pandas as pd
3 import time, os, csv
4 from py2neo import Graph, Node
5
6 #config
7 sample = True
8 path = 'C:\\Temp\\Import\\'
9
10 def LoadLog(localFile):
11     datasetList = []
12     headerCSV = []
13     i = 0
14     with open(localFile) as f:
15         reader = csv.reader(f)
16         for row in reader:
17             if (i==0):
18                 headerCSV = list(row)
19                 i +=1
20             else:
21                 datasetList.append(row)
22
23     log = pd.DataFrame(datasetList, columns=headerCSV)
24
```

```
25      return headerCSV, log
26
27  def CreateEventQuery(logHeader, fileName, LogID = ""):
28      query = f'USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM \"file:///{fileName
        }\" as line'
29      brClose = '}'
30      brOpen = '{'
31      for col in logHeader:
32          if col == 'idx':
33              column = f'toInt(line.{col})'
34          elif col in ['timestamp','start','end']:
35              column = f'datetime(line.{col})'
36          else:
37              column = 'line.'+col
38          newLine = ''
39          if (logHeader.index(col) == 0 and LogID != ""):
40              newLine = f' CREATE (e:Event {brOpen}Log: "{LogID}",{col}: {column},'
41          elif (logHeader.index(col) == 0):
42              newLine = f' CREATE (e:Event {brOpen}{col}: {column},'
43          else:
44              newLine = f' {col}: {column},'
45          if (logHeader.index(col) == len(logHeader)-1):
46              newLine = f' {col}: {column}{brClose})'
47
48          query = query + newLine
49      return query;
50
51  ################### data prep ###################
52
53  def CreateBPI16(path,fileName,sample):
54
55      clicksLog = pd.read_csv(os.path.realpath('BPI2016_Clicks_Logged_In.csv'),
        keep_default_na=True, sep=';',encoding='latin1')
56      complaints = pd.read_csv(os.path.realpath('BPI2016_Complaints.csv'),
        keep_default_na=True, sep=';',encoding='latin1')
57      questions = pd.read_csv(os.path.realpath('BPI2016_Questions.csv'),
        keep_default_na=True, sep=';',encoding='latin1')
58      messages = pd.read_csv(os.path.realpath('BPI2016_Werkmap_Messages.csv'),
        keep_default_na=True, sep=';',encoding='latin1')
59
60      if (sample == True):
61          sampleIds = [2026796, 2223803, 2023026, 114939, 2011721, 2022933, 919259,
        2079086, 466152, 2057965, 1039204, 395673, 1710155, 2081135, 1723340, 1893155,
        1042998, 435939, 1735039, 2045407]
62      else:
63          sampleIds = clicksLog.CustomerID.unique().tolist() # create a list of all
        cases in the dataset
64
65      csvLog = complaints
66      fileNameTmp = fileName+'Complaints.csv'
67      sampleList = [] #create a list (of lists) for the sample data containing a list
         of events for each of the selected cases
68      for case in sampleIds:
69          for index, row in csvLog[csvLog.CustomerID == case].iterrows(): #second
        iteration through the cases for adding data
70              rowList = list(row) #add the event data to rowList
71              sampleList.append(rowList) #add the extended, single row to the sample
        dataset
72
73      header =  list(csvLog) #save the updated header data
74      logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
        and add the samples
75      logSamples.fillna(0)
76      logSamples['ContactDate'] = pd.to_datetime(logSamples['ContactDate'], format='%
        Y-%m-%d')
77      logSamples['ContactDate'] = logSamples['ContactDate'].map(lambda x: x.strftime(
        '%Y-%m-%dT%H:%M:%S.%f')[0:-3]+'+0100')
```

```
78      logSamples = logSamples.rename(columns={'ContactDate': 'timestamp'})
79
80      logSamples.to_csv(path+fileNameTmp, index=True, index_label="idx",na_rep="
        Unknown")
81      logSamples['idx'] = logSamples.index
82
83      complaints = logSamples
84
85      csvLog = questions
86      fileNameTmp = fileName+'Questions.csv'
87      sampleList = [] #create a list (of lists) for the sample data containing a list
         of events for each of the selected cases
88      for case in sampleIds:
89          for index, row in csvLog[csvLog.CustomerID == case].iterrows(): #second
        iteration through the cases for adding data
90              rowList = list(row) #add the event data to rowList
91              sampleList.append(rowList) #add the extended, single row to the sample
        dataset
92
93      header = list(csvLog) #save the updated header data
94      logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
        and add the samples
95      logSamples.fillna(0)
96      logSamples['ContactDate'] = pd.to_datetime(logSamples['ContactDate'], format='%
        Y-%m-%d')
97      logSamples['ContactDate'] = logSamples['ContactDate'].map(lambda x: x.strftime(
        '%Y-%m-%d'))
98      logSamples['ContactTimeStart'] = pd.to_datetime(logSamples['ContactTimeStart'],
         format='%H:%M:%S.%f')
99      logSamples['ContactTimeStart'] = logSamples['ContactTimeStart'].map(lambda x: x
        .strftime('%H:%M:%S.%f')[0:-3])
100     logSamples['start'] = logSamples['ContactDate']+"T"+ logSamples['
        ContactTimeStart'] +'+0100'
101     logSamples['ContactTimeEnd'] = pd.to_datetime(logSamples['ContactTimeEnd'],
        format='%H:%M:%S.%f')
102     logSamples['ContactTimeEnd'] = logSamples['ContactTimeEnd'].map(lambda x: x.
        strftime('%H:%M:%S.%f')[0:-3])
103     logSamples['end'] = logSamples['ContactDate']+"T"+ logSamples['ContactTimeEnd'
        ] +'+0100'
104     logSamples.to_csv(path+fileNameTmp, index=True, index_label="idx",na_rep="
        Unknown")
105     logSamples['idx'] = logSamples.index
106
107     questions = logSamples
108     csvLog = messages
109     fileNameTmp = fileName+'Messages.csv'
110     sampleList = [] #create a list (of lists) for the sample data containing a list
         of events for each of the selected cases
111     for case in sampleIds:
112         for index, row in csvLog[csvLog.CustomerID == case].iterrows(): #second
        iteration through the cases for adding data
113             rowList = list(row) #add the event data to rowList
114             sampleList.append(rowList) #add the extended, single row to the sample
        dataset
115
116     header = list(csvLog) #save the updated header data
117     logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
        and add the samples
118     logSamples.fillna(0)
119     logSamples['EventDateTime'] = pd.to_datetime(logSamples['EventDateTime'],
        format='%Y-%m-%d %H:%M:%S.%f')
120     logSamples['EventDateTime'] = logSamples['EventDateTime'].map(lambda x: x.
        strftime('%Y-%m-%dT%H:%M:%S.%f')[0:-3]+'+0100')
121     logSamples = logSamples.rename(columns={'EventDateTime': 'timestamp'})
122     logSamples['idx'] = range(1, len(logSamples) + 1)
123
124     logSamples['MessageID'] = logSamples['idx'].astype(str) #add prefix to entity
```

```
          ids
125
126       logSamples.to_csv(path+fileNameTmp)
127
128       messages = logSamples
129       csvLog = clicksLog
130       fileNameTmp = fileName+'Clicks.csv'
131       sampleList = [] #create a list (of lists) for the sample data containing a list
           of events for each of the selected cases
132       for case in sampleIds:
133           for index, row in csvLog[csvLog.CustomerID == case].iterrows(): #second
       iteration through the cases for adding data
134               rowList = list(row) #add the event data to rowList
135               sampleList.append(rowList) #add the extended, single row to the sample
       dataset
136
137       header = list(csvLog) #save the updated header data
138       logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
       and add the samples
139       logSamples.fillna(0)
140       logSamples['TIMESTAMP'] = pd.to_datetime(logSamples['TIMESTAMP'], format='%Y-%m
       -%d %H:%M:%S.%f')
141       logSamples['TIMESTAMP'] = logSamples['TIMESTAMP'].map(lambda x: x.strftime('%Y
       -%m-%dT%H:%M:%S.%f')[0:-3]+'+0100')
142       logSamples = logSamples.rename(columns={'TIMESTAMP': 'timestamp', 'PAGE_NAME':
       'Activity'})
143       logSamples = logSamples.drop(logSamples.columns[[range(-1,-10,-1)]], axis=1)
144
145       logSamples.to_csv(path+fileNameTmp, index=True, index_label="idx")
146       logSamples['idx'] = logSamples.index
147
148       clicksLog = logSamples
149
150       return clicksLog, complaints, questions, messages
151
152 ################### import script starts here ###################
153
154 if(sample):
155       fileName = 'BPIC16sample.csv'
156       perfFileName = 'BPIC16samplePerformance.csv'
157 else:
158       fileName = 'BPIC16full.csv'
159       perfFileName = 'BPIC16fullPerformance.csv'
160
161 clicksLog, complaints, questions, messages = CreateBPI16(path,fileName,sample)
162
163 perf = pd.DataFrame(columns=['name', 'start', 'end', 'duration'])
164
165 cbClose = "}"
166 cbOpen = "{"
167
168 Graph = Graph(password="1234")
169 Graph.delete_all() #make sure the neo4j DB is empty
170
171 globalStart = time.time()
172
173 Graph.run('CREATE CONSTRAINT ON (e:Event) ASSERT e.ID IS UNIQUE;') #for
       implementation only (not required by schema or patterns)
174 Graph.run('CREATE CONSTRAINT ON (en:Entity) ASSERT en.uID IS UNIQUE;') #required by
         core pattern
175 Graph.run('CREATE CONSTRAINT ON (l:Log) ASSERT l.ID IS UNIQUE;') #required by core
       pattern
176
177 ##########################################################
178 ######################### sessions(clicks) #############
179 ##########################################################
180
```

A Schema Framework for Graph Event Data

```
181  start = time.time()
182  entities = [['CustomerID', 'Customer'],['Office_U', 'Office_U'],['Office_W', '
         Office_W'],['SessionID', 'Session'],['IPID', 'IP']]
183  startTimestamp = "timestamp"
184  endTimestamp = "timestamp"
185  logID = "Clicks"
186  fileNameTmp = fileName+logID+'.csv'
187  header = list(clicksLog)
188
189  print(f'##################### {logID} import ########################')
190
191  qCreateEvents = CreateEventQuery(header, fileNameTmp,logID) #generate query to
         create all events with all log columns as properties
192  Graph.run(qCreateEvents)
193
194  #no resources per event in the log
195  #create log node and :L_E relationships
196  Graph.create(Node("Log", ID=logID))
197  Graph.run(f'MATCH (e:Event {cbOpen}Log: "{logID}"{cbClose}) MATCH (l:Log {cbOpen}ID
         : "{logID}"{cbClose}) CREATE (l)-[:L_E]->(e)')
198
199
200  for entity in entities: #per entity
201
202      print(entity[0], entity[1])
203
204      #create entity nodes
205      query=f'''MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}"
206      WITH e.{entity[0]} AS id
207      MERGE (en:Entity {cbOpen}ID:("{logID}"+toString(id)) {cbClose})
208      ON CREATE SET en.IDraw = id, en.uID = "{entity[1]}"+"{logID}"+toString(id), en
         .Log = "{logID}", en.EntityType = "{entity[1]}" '''
209      Graph.run(query)
210      print(f'{entity[1]} entity nodes done')
211
212      #create :E_EN relationships
213      query=f'MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}" MATCH (n:Entity
          {cbOpen}EntityType: "{entity[1]}", Log: "{logID}"{cbClose}) WHERE e.{entity
         [0]} = n.IDraw  CREATE (e)-[:E_EN]->(n)'
214      Graph.run(query)
215      print(f'{entity[1]} E_EN relationships done')
216
217      #get all events per entity and add entity-specific index as property
218      query = f'MATCH p = (ev:Event {cbOpen}Log: "{logID}"{cbClose}) -[:E_EN]-> (en:
         Entity {cbOpen}EntityType: "{entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{
         entity[0]}, ev.idx'
219      output = Graph.run(query).data()
220      entityIdx = 0
221      propertyName = f'{entity[1]}_idx'
222      for node in output:
223          node['ev'][propertyName] = entityIdx
224          Graph.push(node['ev'])
225          entityIdx += 1
226      print(f'{entity[1]} internal index added to nodes')
227
228      #create DF relations
229      query = f'''MATCH (l:Log)-[:L_E]->(e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}
         EntityType: "{entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)<-[:L_E]-(l:Log)
230      WHERE e2.{propertyName} - e1.{propertyName} = 1 AND l.ID = "{logID}"
231      MERGE (e1) -[df:DF]-> (e2)
232      ON CREATE SET df.EntityTypes = ["{entity[1]}"]
233      ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
         .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
234      '''
235      Graph.run(query)
236      print(f'{entity[1]} DF relationships done')
237
```

```
238      #create HOW relations
239      #not needed − no resources
240
241  end = time.time()
242  print("Import of the sessions graph took: "+str((end − start))+" seconds.\n")
243
244  perf = perf.append({'name':logID, 'start':start, 'end':end, 'duration':(end − start
         )},ignore_index=True)
245
246  ##############################################################
247  ####################### complaints ###################
248  ##############################################################
249  start = time.time()
250
251  startTimestamp = "timestamp"
252  endTimestamp = "timestamp"
253  logID = "Complaints"
254
255  print(f'##################### {logID} import #########################')
256
257  entities = [['CustomerID', 'Customer'],['Office_U', 'Office_U'],['Office_W', '
         Office_W'],['ComplaintDossierID', 'ComplaintDossier'],['ComplaintID', '
         Complaint']]
258  fileNameTmp = fileName+logID+'.csv'
259  header = list(complaints)
260  qCreateEvents = CreateEventQuery(header, fileNameTmp, logID) #generate query to
         create all events with all log columns as properties
261  Graph.run(qCreateEvents)
262
263  #no resources per event in the log
264
265  #create log node and :L_E relationships
266  Graph.create(Node("Log", ID=logID))
267  Graph.run(f'MATCH (e:Event {cbOpen}Log: "{logID}"{cbClose}) MATCH (l:Log {cbOpen}ID
         : "{logID}"{cbClose}) CREATE (l)−[:L_E]−>(e)')
268
269
270  for entity in entities: #per entity
271
272      print(entity[0], entity[1])
273
274      #create entity nodes
275      query=f'''MATCH (e:Event) <−[:L_E]−(l:Log) WHERE l.ID = "{logID}"
276      WITH e.{entity[0]} AS id
277      MERGE (en:Entity {cbOpen}ID:("{logID}"+toString(id)) {cbClose})
278      ON CREATE SET en.IDraw = id, en.uID = "{entity[1]}"+"{logID}"+toString(id), en
         .Log = "{logID}", en.EntityType = "{entity[1]}" '''
279      Graph.run(query)
280      print(f'{entity[1]} entity nodes done')
281
282      #create :E_EN relationships
283      query=f'MATCH (e:Event) <−[:L_E]−(l:Log) WHERE l.ID = "{logID}" MATCH (n:Entity
          {cbOpen}EntityType: "{entity[1]}", Log: "{logID}"{cbClose}) WHERE e.{entity
         [0]} = n.IDraw   CREATE (e)−[:E_EN]−>(n)'
284      Graph.run(query)
285      print(f'{entity[1]} E_EN relationships done')
286
287      #get all events per entity and add entity−specific index as property
288      query = f'MATCH p = (ev:Event {cbOpen}Log: "{logID}"{cbClose}) −[:E_EN]−> (en:
         Entity {cbOpen}EntityType: "{entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{
         entity[0]}, ev.idx'
289      output = Graph.run(query).data()
290      entityIdx = 0
291      propertyName = f'{entity[1]}_idx'
292      for node in output:
293          node['ev'][propertyName] = entityIdx
294          Graph.push(node['ev'])
```

```
295            entityIdx += 1
296        print(f'{entity[1]} internal index added to nodes')
297
298        #create DF relations
299        query = f'''MATCH (l:Log)-[:L_E]->(e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}
           EntityType: "{entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)<-[:L_E]-(l:Log)
300        WHERE e2.{propertyName} - e1.{propertyName} = 1 AND l.ID = "{logID}"
301        MERGE (e1) -[df:DF]-> (e2)
302        ON CREATE SET df.EntityTypes = ["{entity[1]}"]
303        ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
           .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
304        '''
305        Graph.run(query)
306        print(f'{entity[1]} DF relationships done')
307
308        #create HOW relations
309        #not needed - no resources
310
311  end = time.time()
312  print("Import of the complaints graph took: "+str((end - start))+" seconds.\n")
313
314  perf = perf.append({'name':logID, 'start':start, 'end':end, 'duration':(end - start
       )},ignore_index=True)
315
316  ###############################################################
317  ######################### questions ####################
318  ###############################################################
319  start = time.time()
320
321  startTimestamp = "start"
322  endTimestamp = "end"
323  logID = "Questions"
324
325  print(f'##################### {logID} import ########################')
326
327  entities = [['CustomerID', 'Customer'],['Office_U', 'Office_U'],['Office_W', '
       Office_W'],['QuestionThemeID', 'QuestionTheme'],['QuestionSubthemeID', '
       QuestionSubtheme'],['QuestionTopicID', 'QuestionTopic']]
328  fileNameTmp = fileName+logID+'.csv'
329  header = list(questions)
330  qCreateEvents = CreateEventQuery(header, fileNameTmp, logID) #generate query to
       create all events with all log columns as properties
331  Graph.run(qCreateEvents)
332
333  #no resources per event in the log
334
335  #create log node and :L_E relationships
336  Graph.create(Node("Log", ID=logID))
337  Graph.run(f'MATCH (e:Event {cbOpen}Log: "{logID}"{cbClose}) MATCH (l:Log {cbOpen}ID
       : "{logID}"{cbClose}) CREATE (l)-[:L_E]->(e)')
338
339  for entity in entities: #per entity
340
341        print(entity[0], entity[1])
342
343        #create entity nodes
344        query=f'''MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}"
345        WITH e.{entity[0]} AS id
346        MERGE (en:Entity {cbOpen}ID:("{logID}"+toString(id)) {cbClose})
347        ON CREATE SET en.IDraw = id, en.uID = "{entity[1]}"+"{logID}"+toString(id), en
           .Log = "{logID}", en.EntityType = "{entity[1]}" '''
348        Graph.run(query)
349        print(f'{entity[1]} entity nodes done')
350
351        #create :E_EN relationships
352        query=f'MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}" MATCH (n:Entity
            {cbOpen}EntityType: "{entity[1]}", Log: "{logID}"{cbClose}) WHERE e.{entity
```

```
         [0]} = n.IDraw   CREATE (e)−[:E_EN]−>(n)'
353      Graph.run(query)
354      print(f'{entity[1]} E_EN relationships done')
355
356      #get all events per entity and add entity−specific index as property
357      query = f'MATCH p = (ev:Event {cbOpen}Log: "{logID}"{cbClose}) −[:E_EN]−> (en:
         Entity {cbOpen}EntityType: "{entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{
         entity[0]}, ev.idx'
358      output = Graph.run(query).data()
359      entityIdx = 0
360      propertyName = f'{entity[1]}_idx'
361      for node in output:
362          node['ev'][propertyName] = entityIdx
363          Graph.push(node['ev'])
364          entityIdx += 1
365      print(f'{entity[1]} internal index added to nodes')
366
367      #create DF relations
368      query = f'''MATCH (l:Log)−[:L_E]−>(e1:Event) −[:E_EN]−> (ent:Entity {cbOpen}
         EntityType: "{entity[1]}"{cbClose}) <−[:E_EN]− (e2:Event)<−[:L_E]−(l:Log)
369      WHERE e2.{propertyName} − e1.{propertyName} = 1 AND l.ID = "{logID}"
370      MERGE (e1) −[df:DF]−> (e2)
371      ON CREATE SET df.EntityTypes = ["{entity[1]}"]
372      ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
         .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
373      '''
374      Graph.run(query)
375      print(f'{entity[1]} DF relationships done')
376
377      #create HOW relations
378      #not needed − no resources
379
380  end = time.time()
381  print("Import of the questions graph took: "+str((end − start))+" seconds.\n")
382
383  perf = perf.append({'name':logID, 'start':start, 'end':end, 'duration':(end − start
         )},ignore_index=True)
384
385  ############################################################
386  ######################### messages #########################
387  ############################################################
388  start = time.time()
389
390  startTimestamp = "timestamp"
391  endTimestamp = "timestamp"
392  logID = "Messages"
393
394  print(f'##################### {logID} import ########################')
395
396  entities = [['CustomerID', 'Customer'],['Office_U', 'Office_U'],['Office_W', '
         Office_W'],['MessageID', 'Message'],['HandlingChannelID', 'HandlingChannel']]
397  fileNameTmp = fileName+logID+'.csv'
398  header = list(messages)
399  qCreateEvents = CreateEventQuery(header, fileNameTmp, logID) #generate query to
         create all events with all log columns as properties
400  Graph.run(qCreateEvents)
401
402  #no resources per event in the log
403
404  #create log node and :L_E relationships
405  Graph.create(Node("Log", ID=logID))
406  Graph.run(f'MATCH (e:Event {cbOpen}Log: "{logID}"{cbClose}) MATCH (l:Log {cbOpen}ID
         : "{logID}"{cbClose}) CREATE (l)−[:L_E]−>(e)')
407
408  for entity in entities: #per entity
409
410      print(entity[0], entity[1])
```

```
411
412    #create entity nodes
413    query=f'''MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}"
414    WITH e.{entity[0]} AS id
415    MERGE (en:Entity {cbOpen}ID:("{logID}"+toString(id)) {cbClose})
416    ON CREATE SET en.IDraw = id, en.uID = "{entity[1]}"+"{logID}"+toString(id), en
       .Log = "{logID}", en.EntityType = "{entity[1]}" '''
417    Graph.run(query)
418    print(f'{entity[1]} entity nodes done')
419
420    #create :E_EN relationships
421    query=f'MATCH (e:Event) <-[:L_E]-(l:Log) WHERE l.ID = "{logID}" MATCH (n:Entity
        {cbOpen}EntityType: "{entity[1]}", Log: "{logID}"{cbClose}) WHERE e.{entity
       [0]} = n.IDraw    CREATE (e)-[:E_EN]->(n)'
422    Graph.run(query)
423    print(f'{entity[1]} E_EN relationships done')
424
425    #get all events per entity and add entity-specific index as property
426    query = f'MATCH p = (ev:Event {cbOpen}Log: "{logID}"{cbClose}) -[:E_EN]-> (en:
       Entity {cbOpen}EntityType: "{entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{
       entity[0]}, ev.idx'
427    output = Graph.run(query).data()
428    entityIdx = 0
429    propertyName = f'{entity[1]}_idx'
430    for node in output:
431        node['ev'][propertyName] = entityIdx
432        Graph.push(node['ev'])
433        entityIdx += 1
434    print(f'{entity[1]} internal index added to nodes')
435
436    #create DF relations
437    query = f'''MATCH (l:Log)-[:L_E]->(e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}
       EntityType: "{entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)<-[:L_E]-(l:Log)
438    WHERE e2.{propertyName} - e1.{propertyName} = 1 AND l.ID = "{logID}"
439    MERGE (e1) -[df:DF]-> (e2)
440    ON CREATE SET df.EntityTypes = ["{entity[1]}"]
441    ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
       .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
442    '''
443    Graph.run(query)
444    print(f'{entity[1]} DF relationships done')
445
446    #create HOW relations
447    #not needed - no resources
448
449 end = time.time()
450 print("Import of the questions graph took: "+str((end - start))+" seconds.\n")
451
452 perf = perf.append({'name':logID, 'start':start, 'end':end, 'duration':(end - start
       )},ignore_index=True)
453
454 ###### en_coincide
455
456 #create collector nodes for coinciding entities with log = "none" and create
       relationships to the entities respectively
457 query = '''match (en:Entity) where en.Log = 'Clicks'
458 match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
       and en2.Log = 'Complaints'
459 merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
       .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
460 merge (en)-[:EN_COINCIDE]->(collector)
461 merge (en2)-[:EN_COINCIDE]->(collector)'''
462 Graph.run(query)
463
464 query = '''match (en:Entity) where en.Log = 'Clicks'
465 match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
       and en2.Log = 'Questions'
```

```
466  merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
        .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
467  merge (en)-[:EN_COINCIDE]->(collector)
468  merge (en2)-[:EN_COINCIDE]->(collector)'''
469  Graph.run(query)
470
471  query = '''match (en:Entity) where en.Log = 'Clicks'
472  match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
        and en2.Log = 'Messages'
473  merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
        .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
474  merge (en)-[:EN_COINCIDE]->(collector)
475  merge (en2)-[:EN_COINCIDE]->(collector)'''
476  Graph.run(query)
477
478  query = '''match (en:Entity) where en.Log = 'Complaints'
479  match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
        and en2.Log = 'Questions'
480  merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
        .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
481  merge (en)-[:EN_COINCIDE]->(collector)
482  merge (en2)-[:EN_COINCIDE]->(collector)'''
483  Graph.run(query)
484
485  query = '''match (en:Entity) where en.Log = 'Complaints'
486  match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
        and en2.Log = 'Messages'
487  merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
        .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
488  merge (en)-[:EN_COINCIDE]->(collector)
489  merge (en2)-[:EN_COINCIDE]->(collector)'''
490  Graph.run(query)
491
492  query = '''match (en:Entity) where en.Log = 'Questions'
493  match (en2:Entity) where en.EntityType = en2.EntityType and en.IDraw = en2.IDraw
        and en2.Log = 'Messages'
494  merge (collector:Entity {IDraw: en.IDraw, EntityType: en.EntityType, ID: ("None"+en
        .IDraw), uID: (en.EntityType+"None"+en.IDraw), Log: "None"})
495  merge (en)-[:EN_COINCIDE]->(collector)
496  merge (en2)-[:EN_COINCIDE]->(collector)'''
497  Graph.run(query)
498
499  globalEnd = time.time()
500  print("Total import time: "+str((globalEnd - globalStart))+" seconds.\n")
501  perf = perf.append({'name':'Total', 'start':start, 'end':end, 'duration':(globalEnd
        - globalStart)},ignore_index=True)
502
503  perf.to_csv(perfFileName)
```

A.5.2 Schema Definitions

```
1   pattern_bpic16 = (
2       {//element types
3           Event {Activity, Timestamp}
4           Entity {ID, EntityType, ID+EntityType},
5           Log {ID},
6           E_EN {},
7           L_E {},
8           DF {EntityTypes},
9           EN_COINCIDE {}
10      }
11      {//node types
12          (:Event), (:Entity), (:Log)
13      }
14      {//relationship types
15          (:Event)-[:E_EN]->(:Entity),
16          (:Log)-[:L_E]->(:Event),
```

```
17          ( : Event )−[:DF]−>(:Event ) ,
18          ( : Event )−[:EN_COINCIDE]−>(:Event )
19      }
20      {//inherited patterns
21            0_core , 1_df , 1_en_coincide
22      }
23  )
```

Listing A.5: BPIC 16 Pattern Definition

```
1  schema_bpic16 = (
2      {//element types
3            Event {
4                  Activity !: STRING,
5                  Timestamp !: TIMESTAMP,
6                  start : TIMESTAMP,
7                  end : TIMESTAMP,
8                  CustomerID : INTEGER,
9                  AgeCategory : FLOAT,
10                 Gender : INTEGER,
11                 Office_U : INTEGER,
12                 Office_W : INTEGER,
13                 SessionID : INTEGER,
14                 IPID : INTEGER,
15                 VHOST: STRING,
16                 URL_FILE : STRING,
17                 ComplaintDossierID : STRING,
18                 ComplaintID : STRING,
19                 ContactChannelID : STRING,
20                 ComplaintThemeID : STRING,
21                 ComplaintSubthemeID : STRING,
22                 ComplaintTopicID : STRING,
23                 ComplaintTheme : STRING,
24                 ComplaintSubtheme : STRING,
25                 ComplaintTopic : STRING,
26                 ComplaintTheme_EN : STRING,
27                 ComplaintSubtheme_EN : STRING,
28                 ComplaintTopic_EN : STRING,
29                 EventType : STRING,
30                 HandlingChannelID : INTEGER,
31                 MessageID : INTEGER,
32                 ContactDate : TIMESTAMP,
33                 ContactTimeStart : TIMESTAMP,
34                 ContactTimeEnd : TIMESTAMP,
35                 QuestionThemeID : STRING,
36                 QuestionSubthemeID : STRING,
37                 QuestionTopicID : STRING,
38                 QuestionTheme : STRING,
39                 QuestionSubtheme : STRING,
40                 QuestionTopic : STRING,
41                 QuestionTheme_EN : STRING,
42                 QuestionSubtheme_EN : STRING,
43                 QuestionTopic_EN : STRING
44           }
45           Entity {ID: STRING, EntityType : STRING, ID+EntityType : STRING},
46           Log {ID}: STRING}
47           E_EN {},
48           L_E {},
49           DF {EntityTypes : LIST},
50           HOW {EntityTypes : LIST}
51      }
52      {//node types
53           ( : Event ) , ( : Entity ) , ( : Log )
54      }
55      {//relationship types
56           ( : Event )−[:E_EN]−>(:Entity ) ,
57           ( : Log )−[:L_E]−>(:Event ) ,
```

```
58              (:Event)-[:DF]->(:Event),
59              (:Entity)-[:HOW]->(:Entity)
60      }
61  )
```

<div align="center">Listing A.6: BPIC 16 Schema Definition</div>

A.6 BPIC17

A.6.1 Script Template

```python
1  #loan application
2
3  import pandas as pd
4  import time, os, csv
5  from py2neo import Graph, Node
6
7  #config
8
9  sample = True
10 createNewFile = True
11
12 path = 'C:\\Temp\\Import\\'
13
14 def LoadLog(localFile):
15     datasetList = []
16     headerCSV = []
17     i = 0
18     with open(localFile) as f:
19         reader = csv.reader(f)
20         for row in reader:
21             if (i==0):
22                 headerCSV = list(row)
23                 i +=1
24             else:
25                 datasetList.append(row)
26
27     log = pd.DataFrame(datasetList,columns=headerCSV)
28
29     return headerCSV, log
30
31 def CreateEventQuery(logHeader, fileName, LogID = ""):
32     query = f'USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM \"file:///{fileName}\" as line'
33     brClose = '}'
34     brOpen = '{'
35     for col in logHeader:
36         if col == 'idx':
37             column = f'toInt(line.{col})'
38         elif col in ['timestamp','start','end']:
39             column = f'datetime(line.{col})'
40         else:
41             column = 'line.'+col
42         newLine = ''
43         if (logHeader.index(col) == 0 and LogID != ""):
44             newLine = f' CREATE (e:Event {brOpen}Log: "{LogID}",{col}: {column},'
45         elif (logHeader.index(col) == 0):
46             newLine = f' CREATE (e:Event {brOpen}{col}: {column},'
47         else:
48             newLine = f' {col}: {column},'
49         if (logHeader.index(col) == len(logHeader)-1):
50             newLine = f' {col}: {column}{brClose})'
51
52         query = query + newLine
53     return query;
54
55 def CreateBPI17(path,fileName,sample):
```

```python
56      csvLog = pd.read_csv(os.path.realpath('BPI_Challenge_2017.csv'),
        keep_default_na=True) #load full log from csv
57      csvLog.drop_duplicates(keep='first', inplace=True) #remove duplicates from the
        dataset
58      csvLog = csvLog.reset_index(drop=True) #renew the index to close gaps of
        removed duplicates
59
60      if (sample == True):
61          sampleIds = ['Application_2045572635',
62              'Application_2014483796',
63              'Application_1973871032',
64              'Application_1389621581',
65              'Application_1564472847',
66              'Application_430577010',
67              'Application_889180637',
68              'Application_1065734594',
69              'Application_681547497',
70              'Application_1020381296',
71              'Application_180427873',
72              'Application_2103964126',
73              'Application_55972649',
74              'Application_1076724533',
75              'Application_1639247005',
76              'Application_1465025013',
77              'Application_1244956957',
78              'Application_1974117177',
79              'Application_797323371',
80              'Application_1631297810']
81      else:
82          sampleIds = csvLog.case.unique().tolist() # create a list of all cases in
        the dataset
83
84      csvLog['startTime'] = pd.to_datetime(csvLog['startTime'], format='%Y/%m/%d %H:%
        M:%S.%f')
85      csvLog['start'] = csvLog['startTime'].map(lambda x: x.strftime('%Y-%m-%dT%H:%M
        :%S.%f')[0:-3]+'+0100')
86      csvLog['completeTime'] = pd.to_datetime(csvLog['completeTime'], format='%Y/%m/%
        d %H:%M:%S.%f')
87      csvLog['end'] = csvLog['completeTime'].map(lambda x: x.strftime('%Y-%m-%dT%H:%M
        :%S.%f')[0:-3]+'+0100')
88      csvLog.drop(columns=['startTime','completeTime'], inplace=True)
89      csvLog = csvLog.rename(columns={'event': 'Activity','org:resource':'resource'})
90
91      sampleList = [] #create a list (of lists) for the sample data containing a list
         of events for each of the selected cases
92      for case in sampleIds:
93          for index, row in csvLog[csvLog.case == case].iterrows(): #second iteration
         through the cases for adding data
94              if row['Activity'] == "O_Create Offer": # this activity belongs to an
        offer but has no offer ID
95                  if csvLog.loc[index+1]['Activity'] == 'O_Created':#if next activity
         is "O_Created" (always directly follows "O_Create Offer" [verified with Disco
        ])
96                      row['OfferID'] = csvLog.loc[index+1]['OfferID'] #assign the
        offerID of the next event (O_Created) to this activity
97              rowList = list(row) #add the event data to rowList
98              sampleList.append(rowList) #add the extended, single row to the sample
        dataset
99
100     header =  list(csvLog) #save the updated header data
101     logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
        and add the samples
102     logSamples.fillna(0)
103     logSamples.to_csv(path+fileName, index=True, index_label="idx",na_rep="Unknown"
        )
104
105 if(sample):
```

```
106      fileName = 'BPIC17sample.csv'
107      perfFileName = 'BPIC17samplePerformance.csv'
108  else:
109      fileName = 'BPIC17full.csv'
110      perfFileName = 'BPIC17fullPerformance.csv'
111
112  if(createNewFile):
113      start = time.time()
114      CreateBPI17(path,fileName,sample)
115      end = time.time()
116      print("Prepared data for import in: "+str((end - start))+" seconds.")
117
118  perf = pd.DataFrame(columns=['name', 'start', 'end', 'duration'])
119
120  header, csvLog = LoadLog(path+fileName)
121
122  entities = [['case', 'Application'],['case', 'Workflow'],['OfferID', 'Offer']]
123
124  cbClose = "}"
125  cbOpen = "{"
126
127  Graph = Graph(password="1234")
128
129  Graph.delete_all()
130
131  ############################################################
132  ######################## BPIC 17 ########################
133  ############################################################
134  start = time.time()
135
136  qCreateEvents = CreateEventQuery(header, fileName, 'BPIC17') #generate query to
            create all events with all log columns as properties
137  Graph.run(qCreateEvents)
138  print('Event nodes done')
139
140  #create unique constraints
141  Graph.run('CREATE CONSTRAINT ON (e:Event) ASSERT e.ID IS UNIQUE;') #for
            implementation only (not required by schema or patterns)
142  Graph.run('CREATE CONSTRAINT ON (en:Entity) ASSERT en.uID IS UNIQUE;') #required by
             core pattern
143  Graph.run('CREATE CONSTRAINT ON (l:Log) ASSERT l.ID IS UNIQUE;') #required by core
            pattern
144
145  #create resource (entity) nodes
146  Graph.run("MATCH (e:Event) MERGE(r:Entity {ID: e.resource}) ON CREATE SET r.uID =
            ('Resource'+toString(e.resource)), r.EntityType = 'Resource'")
147  print('Resource nodes done')
148
149  #create :E_EN relationships for resources
150  Graph.run(f'MATCH (e:Event) MATCH (r:Entity {cbOpen}EntityType: "Resource"{cbClose
            }) WHERE r.ID = e.resource CREATE (e)-[:E_EN]->(r)')
151  print('resource :E_EN relationships done')
152
153  #create log node and :L_E relationships
154  Graph.create(Node("Log", ID='BPIC17'))
155  Graph.run(f'MATCH (e:Event {cbOpen}Log: "BPIC17"{cbClose}) MATCH (l:Log {cbOpen}ID:
            "BPIC17"{cbClose}) CREATE (l)-[:L_E]->(e)')
156  print('Log and :L_E relationships done')
157
158  for entity in entities: #per entity
159
160      startEntity = time.time() #per entity
161
162      #create entity nodes
163      query=f'MATCH (e:Event) WHERE e.EventOrigin = "{entity[1]}" WITH e.{entity[0]}
            AS id  MERGE (en:Entity {cbOpen}ID:id, uID:("{entity[1]}"+toString(id)),
            EntityType:"{entity[1]}"{cbClose})'
```

```python
164         Graph.run(query)
165         print(f'{entity[1]} entity nodes done')
166
167         #create :E_EN relationships
168         query=f'MATCH (e:Event) MATCH (n:Entity {cbOpen}EntityType: "{entity[1]}"{
            cbClose}) WHERE e.{entity[0]} = n.ID AND e.EventOrigin = "{entity[1]}" CREATE (
            e)-[:E_EN]->(n)'
169         Graph.run(query)
170         print(f'{entity[1]} E_EN relationships done')
171
172
173
174         start2 = time.time()
175         #get all events per entity and add entity-specific index as property
176         query = f'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "{
            entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{entity[0]}, ev.idx'
177         output = Graph.run(query).data()
178         entityIdx = 0
179         propertyName = f'{entity[1]}_idx'
180         for node in output:
181             node['ev'][propertyName] = entityIdx
182             Graph.push(node['ev'])
183             entityIdx += 1
184
185         end2 = time.time()
186         print(f"Indexing {entity[1]} nodes took: "+str((end2 - start2))+" seconds.")
187
188         #create DF relations
189         query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{
            entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)
190         WHERE e2.{propertyName} - e1.{propertyName} = 1
191         MERGE (e1) -[df:DF]-> (e2)
192         ON CREATE SET df.EntityTypes = ["{entity[1]}"]
193         ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
            .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
194         '''
195         Graph.run(query)
196         print(f'{entity[1]} DF relationships done')
197
198         #create HOW relations
199         query = f'''MATCH (r1:Entity {cbOpen}EntityType: "Resource"{cbClose}) <-[:E_EN
            ]- (e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType:
            "Resource"{cbClose})
200         WHERE '{entity[1]}' IN rel.EntityTypes
201         MERGE (r1)-[how:HOW]->(r2)
202         ON CREATE SET how.EntityTypes = ["{entity[1]}"]
203         ON MATCH SET how.EntityTypes = CASE WHEN "{entity[1]}" IN how.EntityTypes THEN
            how.EntityTypes ELSE how.EntityTypes + "{entity[1]}" END
204         '''
205         Graph.run(query)
206         print(f'{entity[1]} HOW relationships done')
207
208         endEntity = time.time() #import timer per entity
209         perf = perf.append({'name':("BPIC17"+'-'+entity[1]), 'start':startEntity, 'end'
            :endEntity, 'duration':(endEntity - startEntity)},ignore_index=True)
210
211 ########################### CASE A W AND O
212
213 #create case nodes(as entities) with app,wf and offer entities
214 query=f'MATCH (e:Event) WITH e.case AS id  MERGE (en:Entity {cbOpen}ID:id, uID:("
        Case_AWO"+toString(id)), EntityType:"Case_AWO"{cbClose})'
215 Graph.run(query)
216 #create :E_EN relationships
217 query=f'MATCH (e:Event) MATCH (n:Entity {cbOpen}EntityType: "Case_AWO"{cbClose})
        WHERE e.case = n.ID  CREATE (e)-[:E_EN]->(n)'
218 Graph.run(query)
219
```

```
220 query = f'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "Case_AWO"{
        cbClose}) RETURN ev ORDER BY ev.case, ev.idx'
221 output = Graph.run(query).data()
222 entityIdx = 0
223 propertyName = f'Case_AWO_idx'
224 for node in output:
225     node['ev'][propertyName] = entityIdx
226     Graph.push(node['ev'])
227     entityIdx += 1
228
229 #create DF relations
230 query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "Case_AWO"{
        cbClose}) <-[:E_EN]- (e2:Event)
231 WHERE e2.{propertyName} - e1.{propertyName} = 1
232 MERGE (e1) -[df:DF]-> (e2)
233 ON CREATE SET df.EntityTypes = ["Case_AWO"]
234 ON MATCH SET df.EntityTypes = CASE WHEN "Case_AWO" IN df.EntityTypes THEN df.
        EntityTypes ELSE df.EntityTypes + "Case_AWO" END
235 '''
236 Graph.run(query)
237
238 #create HOW relations
239 query = f'''MATCH (r1:Entity {cbOpen}EntityType: "Resource"{cbClose}) <-[:E_EN]- (
        e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType: "
        Resource"{cbClose})
240 WHERE 'Case_AWO' IN rel.EntityTypes
241 MERGE (r1)-[how:HOW]->(r2)
242 ON CREATE SET how.EntityTypes = ["Case_AWO"]
243 ON MATCH SET how.EntityTypes = CASE WHEN "Case_AWO" IN how.EntityTypes THEN how.
        EntityTypes ELSE how.EntityTypes + "Case_AWO" END
244 '''
245 Graph.run(query)
246 print(f'case AWO entities done')
247
248 ######################### CASE A AND O
249
250 #create case nodes with app and offer entities
251 query=f'MATCH (e:Event) WITH e.case AS id  MERGE (:Entity {cbOpen}ID:id, uID:("
        Case_AO"+toString(id)), EntityType:"Case_AO"{cbClose})'
252 Graph.run(query)
253
254 #create :E_EN relationships
255 query=f'''MATCH (e:Event) -[:E_EN]->(ent)
256 WHERE (ent.EntityType IN ["Offer","Application"])
257 MATCH (n:Entity {cbOpen}EntityType:"Case_AO"{cbClose})
258 WHERE n.ID = e.case
259 CREATE (e) -[:E_EN]->(n) '''
260 Graph.run(query)
261
262 query = f'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "Case_AO"{
        cbClose}) RETURN ev ORDER BY ev.case, ev.idx'
263 output = Graph.run(query).data()
264 entityIdx = 0
265 propertyName = f'Case_AO_idx'
266 for node in output:
267     node['ev'][propertyName] = entityIdx
268     Graph.push(node['ev'])
269     entityIdx += 1
270
271 #create DF relations
272 query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "Case_AO"{
        cbClose}) <-[:E_EN]- (e2:Event)
273 WHERE e2.{propertyName} - e1.{propertyName} = 1
274 MERGE (e1) -[df:DF]-> (e2)
275 ON CREATE SET df.EntityTypes = ["Case_AO"]
276 ON MATCH SET df.EntityTypes = CASE WHEN "Case_AO" IN df.EntityTypes THEN df.
        EntityTypes ELSE df.EntityTypes + "Case_AO" END
```

```
277  '''
278  Graph.run(query)
279
280  #create HOW relations
281  query = f'''MATCH (r1:Entity {cbOpen}EntityType: "Resource"{cbClose}) <-[:E_EN]- (
         e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType: "
         Resource"{cbClose})
282  WHERE 'Case_AO' IN rel.EntityTypes
283  MERGE (r1)-[how:HOW]->(r2)
284  ON CREATE SET how.EntityTypes = ["Case_AO"]
285  ON MATCH SET how.EntityTypes = CASE WHEN "Case_AO" IN how.EntityTypes THEN how.
         EntityTypes ELSE how.EntityTypes + "Case_AO" END
286  '''
287  Graph.run(query)
288
289  print(f'case AO entities done')
290
291  ############################# CASE RESOURCE
292
293  caseName = "Case_R"
294  #create case nodes for resource (case) entity
295  query=f'MATCH (e:Event) WITH e.resource AS id  MERGE (:Entity {cbOpen}ID:id, uID
         :("{caseName}"+toString(id)), EntityType:"{caseName}"{cbClose})'
296  Graph.run(query)
297  #create :E_EN relationships
298  query=f'''MATCH (e:Event) -[:E_EN]->(ent:Entity {cbOpen}EntityType: "Resource"{
         cbClose}) MATCH (n:Entity {cbOpen}EntityType: "{caseName}"{cbClose})
299  WHERE ent.ID = n.ID
300  CREATE (e)-[:E_EN]->(n)'''
301  Graph.run(query)
302
303  query = f'MATCH p = (ev:Event) -[:E_EN]-> (en:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) RETURN ev ORDER BY ev.resource, ev.idx'
304  output = Graph.run(query).data()
305  entityIdx = 0
306  propertyName = f'Case_R_idx'
307  for node in output:
308      node['ev'][propertyName] = entityIdx
309      Graph.push(node['ev'])
310      entityIdx += 1
311
312  #create DF relations
313  query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) <-[:E_EN]- (e2:Event)
314  WHERE e2.{propertyName} - e1.{propertyName} = 1
315  MERGE (e1) -[df:DF]-> (e2)
316  ON CREATE SET df.EntityTypes = ["{caseName}"]
317  ON MATCH SET df.EntityTypes = CASE WHEN "{caseName}" IN df.EntityTypes THEN df.
         EntityTypes ELSE df.EntityTypes + "{caseName}" END
318  '''
319  Graph.run(query)
320
321  print(f'case resources entities done')
322
323  end = time.time()
324  print("Import of the graph took: "+str((end - start))+" seconds.")
325
326  perf = perf.append({'name':'BPIC17', 'start':start, 'end':end, 'duration':(end -
         start)},ignore_index=True)
327
328  perf.to_csv(perfFileName)
```

A.6.2 Schema Definitions

Please refer to listing 7.7 for the schema definition and listing 6.5 for the pattern definition.

A.7   BPIC17 with Split Events

A.7.1   Script Template

```python
1  #loan application with split events
2
3  import pandas as pd
4  import time, os, csv
5  from py2neo import Graph, Node
6
7  #config
8
9  sample = True
10 createNewFile = True
11
12 path = 'C:\\Temp\\Import\\'
13
14 def LoadLog(localFile):
15     datasetList = []
16     headerCSV = []
17     i = 0
18     with open(localFile) as f:
19         reader = csv.reader(f)
20         for row in reader:
21             if (i==0):
22                 headerCSV = list(row)
23                 i +=1
24             else:
25                 datasetList.append(row)
26
27     log = pd.DataFrame(datasetList,columns=headerCSV)
28
29     return headerCSV, log
30
31 def CreateEventQuery(logHeader, fileName, LogID = ""):
32     query = f'USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM \"file:///{fileName}\" as line'
33     brClose = '}'
34     brOpen = '{'
35     for col in logHeader:
36         if col == 'idx':
37             column = f'toInt(line.{col})'
38         elif col in ['timestamp','start','end']:
39             column = f'datetime(line.{col})'
40         else:
41             column = 'line.'+col
42         newLine = ''
43         if (logHeader.index(col) == 0 and LogID != ""):
44             newLine = f' CREATE (e:Event {brOpen}Log: "{LogID}",{col}: {column},'
45         elif (logHeader.index(col) == 0):
46             newLine = f' CREATE (e:Event {brOpen}{col}: {column},'
47         else:
48             newLine = f' {col}: {column},'
49         if (logHeader.index(col) == len(logHeader)-1):
50             newLine = f' {col}: {column}{brClose})'
51
52         query = query + newLine
53     return query;
54
55 def CreateBPI17(path,fileName,sample):
56     csvLog = pd.read_csv(os.path.realpath('BPI_Challenge_2017.csv'),
       keep_default_na=True) #load full log from csv
57     csvLog.drop_duplicates(keep='first', inplace=True) #remove duplicates from the
       dataset
58     csvLog = csvLog.reset_index(drop=True) #renew the index to close gaps of
       removed duplicates
59
```

```python
60      if (sample == True):
61          sampleIds = ['Application_2045572635',
62                  'Application_2014483796',
63                  'Application_1973871032',
64                  'Application_1389621581',
65                  'Application_1564472847',
66                  'Application_430577010',
67                  'Application_889180637',
68                  'Application_1065734594',
69                  'Application_681547497',
70                  'Application_1020381296',
71                  'Application_180427873',
72                  'Application_2103964126',
73                  'Application_55972649',
74                  'Application_1076724533',
75                  'Application_1639247005',
76                  'Application_1465025013',
77                  'Application_1244956957',
78                  'Application_1974117177',
79                  'Application_797323371',
80                  'Application_1631297810']
81      else:
82          sampleIds = csvLog.case.unique().tolist() # create a list of all cases in
        the dataset

83
84      csvLog = csvLog.rename(columns={'event': 'Activity','org:resource':'resource'})
85      csvLog['EventIDraw'] = csvLog['EventID']

86
87      sampleList = [] #create a list (of lists) for the sample data containing a list
         of events for each of the selected cases
88      for case in sampleIds:
89          for index, row in csvLog[csvLog.case == case].iterrows(): #second iteration
         through the cases for adding data
90              if row['Activity'] == "O_Create Offer": # this activity belongs to an
        offer but has no offer ID
91                  if csvLog.loc[index+1]['Activity'] == 'O_Created':#if next activity
         is "O_Created" (always directly follows "O_Create Offer" [verified with Disco
        ])
92                      row['OfferID'] = csvLog.loc[index+1]['OfferID'] #assign the
        offerID of the next event (O_Created) to this activity
93              rowEnd = row.copy()

94
95              row.rename({'startTime':'timestamp'}, inplace=True)
96              rowEnd.rename({'completeTime':'timestamp'}, inplace=True)

97
98              row.drop(labels=['completeTime'], inplace=True)
99              rowEnd.drop(labels=['startTime'], inplace=True)

100
101             row['EventID'] = row['EventID']+"Start"
102             rowEnd['EventID'] = rowEnd['EventID']+"End"

103
104             sampleList.append(list(row)) #add the extended, single row to the
        sample dataset
105             sampleList.append(list(rowEnd))

106
107     header =  ['case',
108                 'Activity',
109                 'timestamp',
110                 'LoanGoal',
111                 'ApplicationType',
112                 'RequestedAmount',
113                 'MonthlyCost',
114                 'resource',
115                 'Accepted',
116                 'EventID',
117                 'OfferID',
118                 'FirstWithdrawalAmount',
```

```
119                    'Action',
120                    'Selected',
121                    'CreditScore',
122                    'NumberOfTerms',
123                    'EventOrigin',
124                    'OfferedAmount',
125                    'EventIDraw']
126
127      logSamples = pd.DataFrame(sampleList,columns=header) #create pandas dataframe
         and add the samples
128
129      logSamples['timestamp'] = pd.to_datetime(logSamples['timestamp'], format='%Y/%m
         /%d %H:%M:%S.%f')
130
131      logSamples.fillna(0)
132      logSamples.sort_values(['case','timestamp'], inplace=True)
133      logSamples['timestamp'] = logSamples['timestamp'].map(lambda x: x.strftime('%Y
         -%m-%dT%H:%M:%S.%f')[0:-3]+'+0100')
134
135      logSamples.to_csv(path+fileName, index=True, index_label="idx",na_rep="Unknown"
         )
136
137      return logSamples
138
139  if(sample):
140      fileName = 'BPIC17sample_split.csv'
141      perfFileName = 'BPIC17sample_splitPerformance.csv'
142  else:
143      fileName = 'BPIC17full_split.csv'
144      perfFileName = 'BPIC17full_splitPerformance.csv'
145
146  if(createNewFile):
147      start = time.time()
148      csvMerged = CreateBPI17(path,fileName,sample)
149      end = time.time()
150      print("Prepared data for import in: "+str((end - start))+" seconds.")
151
152  perf = pd.DataFrame(columns=['name', 'start', 'end', 'duration'])
153
154  header, csvLog = LoadLog(path+fileName)
155
156  entities = [['case', 'Application'],['case', 'Workflow'],['OfferID', 'Offer']] #
         define entities end entity identifiers
157
158  cbClose = "}"
159  cbOpen = "{"
160
161  Graph = Graph(password="1234")
162
163  Graph.delete_all()
164
165  ##############################################################
166  #################### BPIC 17 (split) #################
167  ##############################################################
168  start = time.time()
169
170  qCreateEvents = CreateEventQuery(header, fileName, 'BPIC17') #generate query to
         create all events with all log columns as properties
171  Graph.run(qCreateEvents)
172  print('Event nodes done')
173  end = time.time() #import time of all events
174  perf = perf.append({'name':("All Events Import"), 'start':start, 'end':end, '
         duration':(end - start)},ignore_index=True)
175
176  #create unique constraints
177  Graph.run('CREATE CONSTRAINT ON (e:Event) ASSERT e.ID IS UNIQUE;') #for
         implementation only (not required by schema or patterns)
```

```
178 Graph.run('CREATE CONSTRAINT ON (en:Entity) ASSERT en.uID IS UNIQUE;') #required by
         core pattern
179 Graph.run('CREATE CONSTRAINT ON (l:Log) ASSERT l.ID IS UNIQUE;') #required by core
       pattern
180
181 #create resource (entity) nodes
182 Graph.run("MATCH (e:Event) MERGE(r:Entity {ID: e.resource}) ON CREATE SET r.uID =
       ('Resource'+toString(e.resource)), r.EntityType = 'Resource'")
183 print('Resource nodes done')
184
185 #create :E_EN relationships for resources
186 Graph.run(f'MATCH (e:Event) MATCH (r:Entity {cbOpen}EntityType: "Resource"{cbClose
       }) WHERE r.ID = e.resource CREATE (e)-[:E_EN]->(r)')
187 print('resource :E_EN relationships done')
188
189 startTmp = time.time()
190 #create log node and :L_E relationships
191 Graph.create(Node("Log", ID='BPIC17'))
192 Graph.run(f'MATCH (e:Event {cbOpen}Log: "BPIC17"{cbClose}) MATCH (l:Log {cbOpen}ID:
        "BPIC17"{cbClose}) CREATE (l)-[:L_E]->(e)')
193 print('Log and :L_E relationships done')
194 end = time.time() #import time of all events
195 perf = perf.append({'name':("L to E rels"), 'start':startTmp, 'end':end, 'duration'
       :(end - startTmp)},ignore_index=True)
196
197 for entity in entities: #per entity
198
199     startEntity = time.time() #per entity
200
201     #create entity nodes
202     query=f'MATCH (e:Event) WHERE e.EventOrigin = "{entity[1]}" WITH e.{entity[0]}
       AS id  MERGE (en:Entity {cbOpen}ID:id, uID:("{entity[1]}"+toString(id)),
       EntityType:"{entity[1]}"{cbClose})'
203     Graph.run(query)
204     print(f'{entity[1]} entity nodes done')
205
206     #create :E_EN relationships
207     query=f'MATCH (e:Event) MATCH (n:Entity {cbOpen}EntityType: "{entity[1]}"{
       cbClose}) WHERE e.{entity[0]} = n.ID AND e.EventOrigin = "{entity[1]}" CREATE (
       e)-[:E_EN]->(n)'
208     Graph.run(query)
209     print(f'{entity[1]} E_EN relationships done')
210
211
212
213     start2 = time.time()
214     #get all events per entity and add entity-specific index as property
215     query = f'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "{
       entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{entity[0]}, ev.idx'
216     output = Graph.run(query).data()
217     entityIdx = 0
218     propertyName = f'{entity[1]}_idx'
219     for node in output:
220         node['ev'][propertyName] = entityIdx
221         Graph.push(node['ev'])
222         entityIdx += 1
223
224     end2 = time.time()
225     print(f"Indexing {entity[1]} nodes took: "+str((end2 - start2))+" seconds.")
226
227     #create DF relations
228     query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{
       entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)
229     WHERE e2.{propertyName} - e1.{propertyName} = 1
230     MERGE (e1) -[df:DF]-> (e2)
231     ON CREATE SET df.EntityTypes = ["{entity[1]}"]
```

```
232      ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
         .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
233      '''
234      Graph.run(query)
235      print(f'{entity[1]} DF relationships done')
236
237      #create HOW relations
238      query = f'''MATCH (r1:Entity {cbOpen}EntityType:"Resource"{cbClose}) <-[:E_EN
         ]- (e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType:
         "Resource"{cbClose})
239      WHERE '{entity[1]}' IN rel.EntityTypes
240      MERGE (r1)-[how:HOW]->(r2)
241      ON CREATE SET how.EntityTypes = ["{entity[1]}"]
242      ON MATCH SET how.EntityTypes = CASE WHEN "{entity[1]}" IN how.EntityTypes THEN
         how.EntityTypes ELSE how.EntityTypes + "{entity[1]}" END
243      '''
244      Graph.run(query)
245      print(f'{entity[1]} HOW relationships done')
246
247      endEntity = time.time() #import timer per entity
248      perf = perf.append({'name':("BPIC17"+'-'+entity[1]), 'start':startEntity, 'end'
         :endEntity, 'duration':(endEntity - startEntity)},ignore_index=True)
249
250  coStart = time.time()
251
252  query = '''match (e:Event) <-[:L_E]-(l:Log) -[:L_E]->(e2:Event)
253  where e.EventIDraw = e2.EventIDraw
254  MERGE (collector:Event {ID: e.EventIDraw})
255  MERGE (e)-[:E_COINCIDE]->(collector)
256  MERGE (e2)-[:E_COINCIDE]->(collector)'''
257  Graph.run(query)
258
259  coEnd = time.time()
260  print("coincides took: "+str((coEnd - coStart))+" seconds.")
261
262
263  end = time.time()
264  print("Import of the graph took: "+str((end - start))+" seconds.")
265
266  perf = perf.append({'name':'BPIC17', 'start':start, 'end':end, 'duration':(end -
         start)},ignore_index=True)
267
268  perf.to_csv(perfFileName)
```

A.7.2   Schema Definitions

Please refer to listing 7.12 for the schema definition and listing 7.11 for the pattern definition.

A.8   BPIC19

A.8.1   Script Template

```
1   import pandas as pd
2   import time, csv, os
3   from py2neo import Graph, Node
4
5   ### config
6
7   sample = True              # False = full data set, True = sample data set
8   createNewFile = True       # Update the CSV file for import?
9
10  path = 'C:\\Temp\\Import\\'
11
12  def LoadLog(localFile):
13      datasetList = []
```

```
14      headerCSV = []
15      i = 0
16      with open(localFile) as f:
17          reader = csv.reader(f)
18          for row in reader:
19              if (i==0):
20                  headerCSV = list(row)
21                  i +=1
22              else:
23                  datasetList.append(row)
24
25      log = pd.DataFrame(datasetList, columns=headerCSV)
26
27      return headerCSV, log
28
29  def CreateEventQuery(logHeader, fileName, LogID = ""):
30      query = f'USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM \"file:///{fileName
        }\" as line'
31      brClose = '}'
32      brOpen = '{'
33      for col in logHeader:
34          if col == 'idx':
35              column = f'toInt(line.{col})'
36          elif col in ['timestamp','start','end']:
37              column = f'datetime(line.{col})'
38          else:
39              column = 'line.'+col
40          newLine = ''
41          if (logHeader.index(col) == 0 and LogID != ""):
42              newLine = f' CREATE (e:Event {brOpen}Log: "{LogID}",{col}: {column},'
43          elif (logHeader.index(col) == 0):
44              newLine = f' CREATE (e:Event {brOpen}{col}: {column},'
45          else:
46              newLine = f' {col}: {column},'
47          if (logHeader.index(col) == len(logHeader)-1):
48              newLine = f' {col}: {column}{brClose})'
49
50          query = query + newLine
51      return query;
52
53  def CreateBPI19(path,fileName,bSample):
54      datasetList = []
55      headerCSV = []
56      i = 0
57      print('Loading source ' + str(time.time()))
58      with open(os.path.realpath('BPI_Challenge_2019.csv')) as f:
59          reader = csv.reader(f)
60          for row in reader:
61              if (i==0):
62                  headerCSV = list(row)
63                  i +=1
64              else:
65                  datasetList.append(row)
66      print('Renaming columns ' + str(time.time()))
67      csvLog = pd.DataFrame(datasetList, columns=headerCSV)
68
69      csvLog.drop(columns=['event User','case Source'], inplace=True) #redundant
70
71      csvLog.rename(columns={'case concept:name':'cID',
72                             'case Purchasing Document':'cPOID',
73                             'eventID ':'ID',
74                             'case Spend area text':'cSpendAreaText',
75                             'case Company':'cCompany',
76                             'case Document Type':'cDocType',
77                             'case Sub spend area text':'cSubSPendAreaText',
78                             'case Purch. Doc. Category name':'cPurDocCat',
79                             'case Vendor':'cVendor',
```

```
80                                'case Item Type':'cItemType',
81                                'case Item Category':'cItemCat',
82                                'case Spend classification text':'cSpendClassText',
83                                'case Name':'cVendorName',
84                                'case GR-Based Inv. Verif.':'cGRbasedInvVerif',
85                                'case Item':'cItem',
86                                'case Goods Receipt':'cGR',
87                                'event org:resource':'resource',
88                                'event concept:name':'activity',
89                                'event Cumulative net worth (EUR)':'eCumNetWorth',
90                                'event time:timestamp':'timestamp'}, inplace=True)
91       print('Changing DateTime format '+ str(time.time()))
92      csvLog['timestamp'] = pd.to_datetime(csvLog['timestamp'], format='%d-%m-%Y %H:%
      M:%S.%f')
93      csvLog['timestamp'] = csvLog['timestamp'].map(lambda x: x.strftime('%Y-%m-%dT%H
      :%M:%S.%f')[0:-3]+'+0100')
94
95      if (bSample == True):
96          sampleIds = ['4508062571',
97            '4507010217',
98            '4507000321',
99            '4507040910',
100           '4507021063',
101           '4507024440',
102           '4507001109',
103           '4507020425',
104           '4507014406',
105           '4507018608',
106           '4508066411',
107           '4508053414',
108           '4507010940',
109           '4507022053',
110           '4507016146',
111           '4508044395',
112           '4508072550',
113           '4507002104',
114           '4507020767',
115           '4508057849']
116
117          dfSize = len(csvLog.index)
118          #PO is defined as case (instead of PO line item)
119          sampleList = [] #create a list (of lists) for the sample data containing a
      list of events for each of the selected cases
120          i = 0
121          for case in sampleIds:
122              for index, row in csvLog[csvLog.cPOID == case].iterrows():
123                  i += 1
124                  rowList = list(row) #add the event data to rowList
125                  sampleList.append(rowList) #add the extended, single row to the
      sample dataset
126              if (i%100 == 0):
127                  print(i+' of '+dfSize+' written...')
128
129          header =  list(csvLog) #save the updated header data
130          logSamples = pd.DataFrame(sampleList,columns=header) #create pandas
      dataframe and add the samples
131          logSamples.to_csv(path+fileName, index=True, index_label="idx",na_rep="
      Unknown")
132
133      else:
134          csvLog.to_csv(path+fileName, index=True, index_label="idx",na_rep="Unknown"
      )
135
136 ### create the graph
137
138 if(sample):
139      fileName = 'BPIC19sample.csv'
```

```
140        perfFileName = 'BPIC19samplePerformance.csv'
141  else:
142        fileName = 'BPIC19full.csv'
143        perfFileName = 'BPIC19fullPerformance.csv'
144
145  if(createNewFile):
146        start = time.time()
147        CreateBPI19(path,fileName,sample)
148        end = time.time()
149        print("Prepared data for import in: "+str((end - start))+" seconds.")
150
151
152  perf = pd.DataFrame(columns=['name', 'start', 'end', 'duration'])
153
154  header, csvLog = LoadLog(path+fileName)
155
156  entities = [['cID', 'POI']]
157
158  cbClose = "}"
159  cbOpen = "{"
160
161  Graph = Graph(password="1234")
162
163  Graph.delete_all()
164
165  ###########################################################
166  ####################### BPIC 19 ########################
167  ###########################################################
168  print('Starting BPIC 19 import')
169  start = time.time()
170
171  #create unique constraints
172  Graph.run('CREATE CONSTRAINT ON (e:Event) ASSERT e.ID IS UNIQUE;') #for
          implementation only (not required by schema or patterns)
173  Graph.run('CREATE CONSTRAINT ON (en:Entity) ASSERT en.uID IS UNIQUE;') #required by
            core pattern
174  Graph.run('CREATE CONSTRAINT ON (l:Log) ASSERT l.ID IS UNIQUE;') #required by core
          pattern
175
176  qCreateEvents = CreateEventQuery(header, fileName, 'BPIC19') #generate query to
          create all events with all log columns as properties
177  Graph.run(qCreateEvents)
178
179  print('Event nodes done')
180
181  #create resource (entity) nodes
182  Graph.run("MATCH (e:Event) MERGE(r:Entity {ID: e.resource}) ON CREATE SET r.uID =
          ('Resource'+toString(e.resource)), r.EntityType = 'Resource'")
183
184  print('Resource nodes done')
185
186  #create :E_EN relationships for resources
187  Graph.run('MATCH (e:Event) MATCH (r:Entity {EntityType: "Resource"}) WHERE r.ID = e
          .resource CREATE (e)-[:E_EN]->(r)')
188
189  print('resource :E_EN relationships done')
190
191  #create log node and :L_E relationships
192  Graph.create(Node("Log", ID='BPIC19'))
193  Graph.run('MATCH (e:Event {Log: "BPIC19"}) MATCH (l:Log {ID: "BPIC19"}) CREATE (l)
          -[:L_E]->(e)')
194
195  print('Log and :L_E relationships done')
196
197  for entity in entities: #per entity
198
199        startEntity = time.time() #per entity
```

```python
200
201     #create entity nodes
202     query=f 'MATCH (e:Event) WITH e.{entity[0]} AS id  MERGE (en:Entity {cbOpen}ID:
        id, uID:("{entity[1]}"+toString(id)), EntityType:"{entity[1]}"{cbClose})'
203     Graph.run(query)
204     print(f'{entity[1]} entity nodes done')
205     #create :E_EN relationships
206     query=f 'MATCH (e:Event) MATCH (n:Entity {cbOpen}EntityType: "{entity[1]}"{
        cbClose}) WHERE e.{entity[0]} = n.ID  CREATE (e)-[:E_EN]->(n)'
207     Graph.run(query)
208     print(f'{entity[1]} E_EN relationships done')
209
210
211     #get all events per entity and add entity-specific index as property
212     query = f 'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "{
        entity[1]}"{cbClose}) RETURN ev ORDER BY ev.{entity[0]}, ev.idx'
213     output = Graph.run(query).data()
214     entityIdx = 0
215     propertyName = f'{entity[1]}_idx'
216     for node in output:
217         node['ev'][propertyName] = entityIdx
218         Graph.push(node['ev'])
219         entityIdx += 1
220     print(f'{entity[1]} internal index added to nodes')
221
222     #create DF relations
223     query = f '''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{
        entity[1]}"{cbClose}) <-[:E_EN]- (e2:Event)
224     WHERE e2.{propertyName} - e1.{propertyName} = 1
225     MERGE (e1) -[df:DF]-> (e2)
226     ON CREATE SET df.EntityTypes = ["{entity[1]}"]
227     ON MATCH SET df.EntityTypes = CASE WHEN "{entity[1]}" IN df.EntityTypes THEN df
        .EntityTypes ELSE df.EntityTypes + "{entity[1]}" END
228     '''
229     Graph.run(query)
230     print(f'{entity[1]} DF relationships done')
231
232     #create HOW relations
233     query = f '''MATCH (r1:Entity {cbOpen}EntityType: "Resource"{cbClose}) <-[:E_EN
        ]- (e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType:
        "Resource"{cbClose})
234     WHERE '{entity[1]}' IN rel.EntityTypes
235     MERGE (r1)-[how:HOW]->(r2)
236     ON CREATE SET how.EntityTypes = ["{entity[1]}"]
237     ON MATCH SET how.EntityTypes = CASE WHEN "{entity[1]}" IN how.EntityTypes THEN
        how.EntityTypes ELSE how.EntityTypes + "{entity[1]}" END
238     '''
239     Graph.run(query)
240
241     print(f'{entity[1]} HOW relationships done')
242
243     endEntity = time.time() #import timer per entity
244     perf = perf.append({'name':("BPIC19"+'-'+entity[1]), 'start':startEntity, 'end'
        :endEntity, 'duration':(endEntity - startEntity)},ignore_index=True)
245
246 ############################## CASE RESOURCE
247
248 caseName = "Case_R"
249 #create case nodes for resource (case) entity
250 query=f 'MATCH (e:Event) WITH e.resource AS id  MERGE (:Entity {cbOpen}ID:id, uID
        :("{caseName}"+toString(id)), EntityType:"{caseName}"{cbClose})'
251 Graph.run(query)
252 #create :E_EN relationships
253 query=f '''MATCH (e:Event)-[:E_EN]->(ent:Entity {cbOpen}EntityType: "Resource"{
        cbClose}) MATCH (n:Entity {cbOpen}EntityType: "{caseName}"{cbClose})
254 WHERE ent.ID = n.ID
255 CREATE (e)-[:E_EN]->(n) '''
```

```
256  Graph.run(query)
257
258  query = f'MATCH p = (ev:Event) -[:E_EN]-> (en:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) RETURN ev ORDER BY ev.resource, ev.idx'
259  output = Graph.run(query).data()
260  entityIdx = 0
261  propertyName = f'{caseName}_idx'
262  for node in output:
263      node['ev'][propertyName] = entityIdx
264      Graph.push(node['ev'])
265      entityIdx += 1
266
267  #create DF relations
268  query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) <-[:E_EN]- (e2:Event)
269  WHERE e2.{propertyName} - e1.{propertyName} = 1
270  MERGE (e1) -[df:DF]-> (e2)
271  ON CREATE SET df.EntityTypes = ["{caseName}"]
272  ON MATCH SET df.EntityTypes = CASE WHEN "{caseName}" IN df.EntityTypes THEN df.
         EntityTypes ELSE df.EntityTypes + "{caseName}" END
273  '''
274  Graph.run(query)
275
276  ############################# Resource Entity w/o "NONE" user
277
278  caseName = "Case_R_NoNone"
279  #create case nodes for resource (case) entity
280  query=f'MATCH (e:Event) WITH e.resource AS id WHERE id <> "NONE" MERGE (:Entity {
         cbOpen}ID:id, uID:("{caseName}"+toString(id)), EntityType:"{caseName}"{cbClose
         })'
281  Graph.run(query)
282  #create :E_EN relationships
283  query=f'''MATCH (e:Event)-[:E_EN]->(ent:Entity {cbOpen}EntityType: "Resource"{
         cbClose}) MATCH (n:Entity {cbOpen}EntityType: "{caseName}"{cbClose})
284  WHERE ent.ID = n.ID
285  CREATE (e)-[:E_EN]->(n)'''
286  Graph.run(query)
287
288  query = f'MATCH p = (ev:Event) -[E_EN]-> (en:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) RETURN ev ORDER BY ev.resource, ev.idx'
289  output = Graph.run(query).data()
290  entityIdx = 0
291  propertyName = f'{caseName}_idx'
292  for node in output:
293      node['ev'][propertyName] = entityIdx
294      Graph.push(node['ev'])
295      entityIdx += 1
296
297  #create DF relations
298  query = f'''MATCH (e1:Event) -[:E_EN]-> (ent:Entity {cbOpen}EntityType: "{caseName
         }"{cbClose}) <-[:E_EN]- (e2:Event)
299  WHERE e2.{propertyName} - e1.{propertyName} = 1
300  MERGE (e1) -[df:DF]-> (e2)
301  ON CREATE SET df.EntityTypes = ["{caseName}"]
302  ON MATCH SET df.EntityTypes = CASE WHEN "{caseName}" IN df.EntityTypes THEN df.
         EntityTypes ELSE df.EntityTypes + "{caseName}" END
303  '''
304  Graph.run(query)
305
306  #create HOW relations
307  query = f'''MATCH (r1:Entity {cbOpen}EntityType: "Resource"{cbClose}) <-[:E_EN]- (
         e1:Event) -[rel:DF]-> (e2:Event) -[:E_EN]-> (r2:Entity {cbOpen}EntityType: "
         Resource"{cbClose})
308  WHERE '{caseName}' IN rel.EntityTypes
309  MERGE (r1)-[how:HOW]->(r2)
310  ON CREATE SET how.EntityTypes = ["{caseName}"]
```

```
311 ON MATCH SET how.EntityTypes = CASE WHEN "{caseName}" IN how.EntityTypes THEN how.
        EntityTypes ELSE how.EntityTypes + "{caseName}" END
312 '''
313 Graph.run(query)
314
315 end = time.time()
316 print("Import of the graph took: "+str((end - start))+" seconds.")
317
318 perf = perf.append({'name':'BPIC19', 'start':start, 'end':end, 'duration':(end -
        start)},ignore_index=True)
319
320 perf.to_csv(perfFileName)
```

A.8.2 Schema Definitions

```
1  pattern_bpic19 = (
2      {//element types
3          Event {Activity, Timestamp}
4          Entity {ID, EntityType, ID+EntityType},
5          Log {ID},
6          E_EN {},
7          L_E {},
8          DF {EntityTypes},
9          HOW {EntityTypes}
10     }
11     {//node types
12         (:Event), (:Entity), (:Log)
13     }
14     {//relationship types
15         (:Event)-[:E_EN]->(:Entity),
16         (:Log)-[:L_E]->(:Event),
17         (:Event)-[:DF]->(:Event),
18         (:Entity)-[:HOW]->(:Entity)
19     }
20     {//inherited patterns
21         0_core, 1_df, 2_how
22     }
23 )
```

Listing A.7: BPIC 19 Pattern Definition

```
1  schema_bpic19 = (
2      {//element types
3          Event {
4              Activity!: STRING,
5              Timestamp!: TIMESTAMP,
6            resource: STRING,
7              ID: INT,
8              cID: INT,
9              cSpendAreaText: STRING,
10             cCompany: STRING,
11             cDocType: STRING,
12             cSubSPendAreaText: STRING,
13             cPOID: INT,
14             cPurDocCat: STRING,
15             cItemType: STRING,
16             cVendor: STRING,
17             cItemCat: STRING,
18             cSpendClassText: STRING,
19             cVendorName: STRING,
20             cGRbasedInvVerif: STRING,
21             cItem: STRING,
22             cGR: BOOLEAN,
23             eCumNetWorth: DOUBLE
24         }
25         Entity {ID: STRING, EntityType: STRING, ID+EntityType: STRING},
26         Log {ID}: STRING}
```

```
27            E_EN  {},
28            L_E  {},
29            DF  {EntityTypes:  LIST},
30            HOW  {EntityTypes:  LIST}
31        }
32    {//node  types
33            (:Event),  (:Entity),  (:Log)
34        }
35    {//relationship  types
36            (:Event)−[:E_EN]−>(:Entity),
37            (:Log)−[:L_E]−>(:Event),
38            (:Event)−[:DF]−>(:Event),
39            (:Entity)−[:HOW]−>(:Entity)
40        }
41 )
```

Listing A.8: BPIC 19 Schema Definition