MASTER

Automated Translation of Event Data from Relational to Graph Databases

Leander, J.

Award date:
2020

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

Automated Translation of Event Data from Relational to Graph Databases

Master Thesis

J. Leander

Supervisors:
Dr. Dirk Fahland
Dr. George Fletcher

Assessment Committee:
Dr. Dirk Fahland
Dr. George Fletcher
Dr. Laura Genga

Eindhoven, May 2020

Abstract

Process data is often multi-dimensional, but is made one-dimensional by transforming it into an event log, as each event in an event log can only have one case identifier. During this transformation we lose interesting relations between entities involved in that process. If we instead transform the process data into a graph representing event data, where every event can have multiple case identifiers, we don't lose this multi-dimensionality and thus keep the interesting relations between entities. In this thesis we discuss how to automatically transform event data from a relational to a graph database with one-to-one, one-to-many and many-to-many relationships between events and case identifiers, such that the transformed data captures both structural and temporal relations in the data. We design a graph data model that is able to capture these structural and temporal relations. We formulate requirements to which input data to this transformation should conform and look into what domain knowledge and which transformation steps are required to perform this transformation. Using this transformation, we can easily transform any relational database that conforms to our requirements into an event data graph, which we successfully did for two real-life datasets.

Contents

Chapter 1

Introduction

Process mining, as described by Wil van der Aalst [11], is a combination of business process management and data mining. Its main purpose is to facilitate finding business insights in event data produced by information systems, usually in the form of an event log.

Such an *event log* is a collection of cases of a process. A *case* is a sequence of events that describes one run of the process. Each *event* has at least a case identifier, a timestamp and an activity name, but can have an arbitrary number of other attributes to further describe the event.

Often, information systems don't directly produce an event log, in which case it must first be built using data from, for instance, multiple tables in a relational database, which together describe a *process*. Often, such a process involves multiple inter-related entities, like employees and information objects, each linked to a number of case identifiers, whereas an event log describes events as if there is only a single case identifier. In essence this means that to build an event log from a relational database, we reduce multi-dimensional data to a single dimension and thus denormalize the data.

Due to the denormalized structure of an event log, interesting information like relations between tables is lost with respect to the normalized event data, that exists in databases before we transform it into an event log. Normalized event data is event data that does not suffer from convergence and divergence [7]. However, this normalized event data is hard to use for process mining algorithms, since it can be structured in many different ways, whereas an event log always uses the same structure where cases consist of a sequence of events, each described by a number of attributes.

A downside of storing event data as relational data, normalized or not, is that it does not directly facilitate queries that are commonly needed in process mining algorithms [6]. For instance, an SQL query to retrieve only cases where event A is directly followed by event B is very complex and an SQL query to retrieve a path of such 'directly follows' relationships between events is impossible to formulate.

Event data can also be stored in a graph database, like Neo4J. We call this data 'event graph data'. However, there is no standard way to do this yet. In a previous Master Thesis [5] and paper[6], Stefan Esser proposed a graph data model for this purpose and transformed several event logs to event graph data that uses this data model, to show that it can be used for common process mining algorithm operations, including 'directly follows' relations. Another upside of graph event data is that events and case identifiers can exist in one-to-one, one-to-many and many-to-many relationships, whereas events and cases can only exist in a one-to-one relation in a classic event log.

In this thesis we explore how to automatically transform event data from relational databases to graph databases. Esser assumed event logs to be the input of the transformation to his event graph data model. He re-normalized data from event logs as good as possible in order to create a multi-dimensional version of that event data. It is, however, often not possible to recreate the data that was used to build the event log, as event logs often do not contain the necessary information to do so. This is why Esser used domain knowledge to create multi-dimensional data from event logs.

However, using a relational database as the starting point for the data transformation introduces additional challenges that are not yet solved by his data model. We propose a semi-automatic approach that can directly transform all existing structural relations and correlations in the data. Though, since event data can be stored in relational databases in different ways, we require some user-provided specification to specify how to extract events from the tables, making the approach semi-automatic.

We therefore try to answer the following research question:

How to automatically transform event data from a relational to a graph database with one-to-one, one-to-many and many-to-many relationships between events and case identifiers, such that the transformed data captures both structural and temporal relations in the data?

1. To what requirements does input data need to conform, so that it can be transformed into graph data capturing both the input data's structural and temporal relations?
2. What are the requirements of a data model to represent normalized event data in a graph database?
3. What are the transformation steps needed to go from relational event data to graph event data?
4. What is the domain knowledge required to perform the transformation?

Figure 1.1 shows an overview of the steps of the transformation we propose in this thesis, along with the sections in which their discussions can be found. This figure can be split into two parts.

The first part is Prior work and Background, which are the components inside the Figure's rectangle. Here we find R2PG-DM [9, 10], which transforms relational data to generic *property graphs*. We also find Esser's data model for *event graphs* with explicit event data [5], which is the data model we will use as a basis for our event graph data model.

The second part, representing our contributions, can be found outside this rectangle. Not every relational database is suitable for our transformation, so a relational database may need to be altered in order to conform to our requirements, which we list in Section 3.1. The first transformation step involves our improved version of R2PG-DM. Our improved version of R2PG-DM works like the original, as explained in Section 2.4, but we identified a number of problems in the original approach of R2PG-DM, which we overcome in Sections 4.1 and 5.1. R2PG-DM's output is not suitable for efficient importation into existing graph database systems, so we propose a transformation, that transforms R2PG-DM's output, such that it can be used for bulk importation, which we describe in Section 5.1.4.

After importing R2PG-DM's output into a graph database, we have a generic property graph describing the records and relations of a relational database in graph-form. We then need to make the concepts of event graphs, i.e. events, entities, and their relations explicit, as explained in Section 3.2. To do this, we first define an *EER model*, which we discuss in Section 4.2.1. then, using a series of transformation steps, discussed in Sections 4.2.2-4.2.5 and 5.2.2-5.2.6, we introduce the event graph concepts to the graph database.

Our improved version of R2PG-DM can be found in this github repository: `https://github.com/jamiro24/R2PG-DM`, The other transformation steps can be found in the following github repository: `https://github.com/jamiro24/Relational-Database-to-Event-Graph`

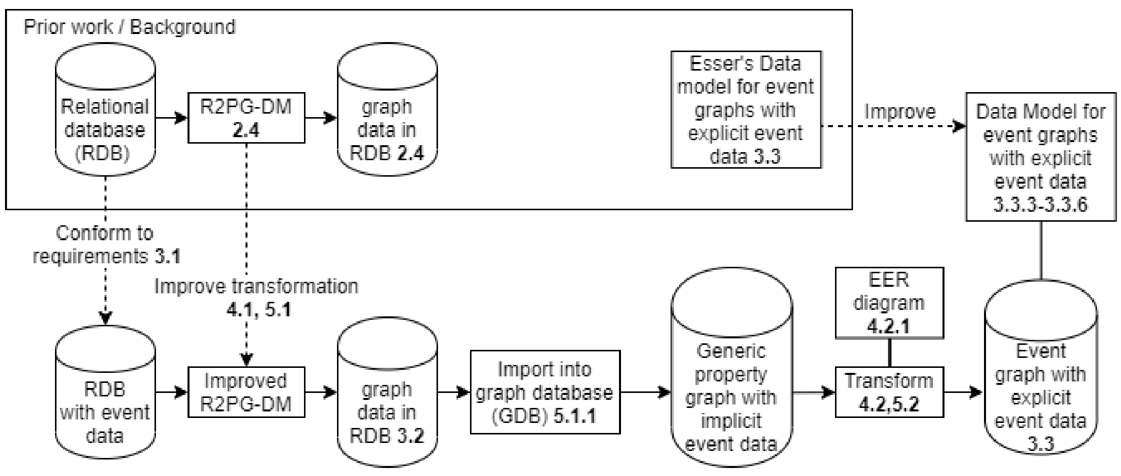
Prior work / Background
Relational database (RDB)
R2PG-DM 2.4
graph data in RDB 2.4
Esser's Data model for event graphs with explicit event data 3.3
Improve
Data Model for event graphs with explicit event data 3.3.3-3.3.6
Conform to requirements 3.1
Improve transformation 4.1, 5.1
RDB with event data
Improved R2PG-DM
graph data in RDB 3.2
Import into graph database (GDB) 5.1.1
Generic property graph with implicit event data
EER diagram 4.2.1
Transform 4.2,5.2
Event graph with explicit event data 3.3

Figure 1.1: Overview of discussed topics and in which sections they can are discussed

Chapter 2

Preliminaries

2.1 Event logs

An event log [11] is a collection of events from a single process, often stored in a *.csv* or *.xes* file. Each event relates to one process instance, which we call the case of that event. Cases are distinguished by a case identifier. An event also needs an activity name, which describes what sort of event it is. Furthermore, the events need to be ordered, for example on a timestamp. The case identifier, activity name and ordering attribute are stored together with the event, which in case of a *.csv* file means in the same row, but separate columns. These requirements form the bare minimum information needed about each event, such that the event log can be used for process mining.

Events can also contain additional information, like resources or costs. We refer to these properties, as well as the minimally required properties, as attributes. These attributes can be further distinguished by whether they are defined at trace or at event level. A trace attribute's value does not change during the runtime of a case, where an event attribute can have a different value during each individual event.

Table 2.1 shows part of an example event log. In this log, the *Case id* column distinguishes process instances. Each event relates to one of these Case ids, as well as to an activity in the *Activity* column. Furthermore they are ordered by the *Event id* column, but they could have been ordered by the *Timestamp* column as well. Finally, they have two additional attributes, in columns *Resource* and *Cost.* These attributes are not necessary for process mining, but could be useful for analysing this process.

Case id	Event id	Timestamp	Activity	Resource	Cost
1	35654423	30-12-2010:11.02	register request	Pete	50
	35654424	31-12-2010:10.06	examine thoroughly	Sue	400
	35654425	05-01-2011:15.12	check ticket	Mike	100
	35654426	06-01-2011:11.18	decide	Sara	200
	35654427	07-01-2011:14.24	reject request	Pete	200
2	35654483	30-12-2010:11.32	register request	Mike	50
	35654485	30-12-2010:12.12	check ticket	Mike	100
	35654487	30-12-2010:14.16	examine casually	Pete	400
	35654488	05-01-2011:11.22	decide	Sara	200
	35654489	08-01-2011:12.05	pay compensation	Ellen	200
...	...	...	...	...	...

Table 2.1: Excerpt from an example event log [11]

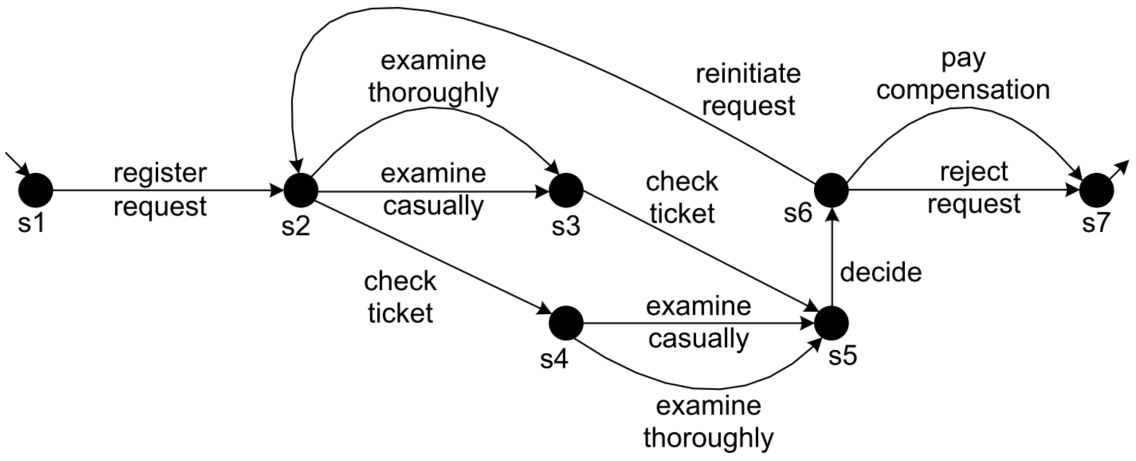

Figure 2.1: Example of a discovered process model [11]

A process mining algorithm can, for example, take such an event log, and use it to discover a process model, or compare it with an existing process model to see whether it fits. Figure 2.1 shows a possible process model for the event log of Table 2.1.

One tool, often academically used for process mining, is called ProM[4]. ProM allows you to run various process mining algorithms on any event log you provide, analyze these event logs, and perform various other operations on these event logs.

2.2 Relational databases

A relational database uses the relational model for database management. Data is represented by tuples, which are grouped into relations. Each relation, often called table, contains a number of columns, each distinguishable by their name. A column is semantically the same as an event log attribute. Each tuple, often called row, consists of a value for each column of their table (possibly a *null* value). Each table has a primary key, consisting of one or more columns, with which each row can be uniquely identified. A table can have zero or more foreign keys, consisting of one or more columns, each indicating a relationship to the primary key of another table. This means that the values of a foreign key's columns in a row, must either be *null* or exist as a primary key in the related table. Case identifier attributes in event logs can be seen as foreign keys for event records, that all refer to the same case in a table that is not part of the event log table.

Relational databases can be structured in numerous ways. Two terms often used in regards to their structure are normalization and denormalization. Normalization is the process of organizing a relational database to reduce data redundancy. The main idea is to organize the data such that each table contains information about one specific topic. A denormalized database contains tables describing multiple topics. This often leads to duplicated data. Denormalization can be used to improve database performance at the cost of storage space [8].

The structure, or schema, of a relational database can be described using an ER diagram. Figure 2.2 shows an example of an ER diagram for the tables shown in Table 2.2 Such an ER diagram lists all the database's tables and shows the names of the columns found in each table. One or more columns can be underlined in the ER diagram to designate the table's primary key, which (together) uniquely identify each row of that table. Furthermore it describes which tables reference which tables via foreign keys, using lines between tables, and shows the multiplicity of those foreign key relations.

The most popular query language for relational databases is called SQL (Structured Query Language). SQL allows you to retrieve, create, modify and delete data. There are numerous variants of SQL, which are all slightly different, but all follow some version of the *ISO/IEC*

9075 [3] standard.

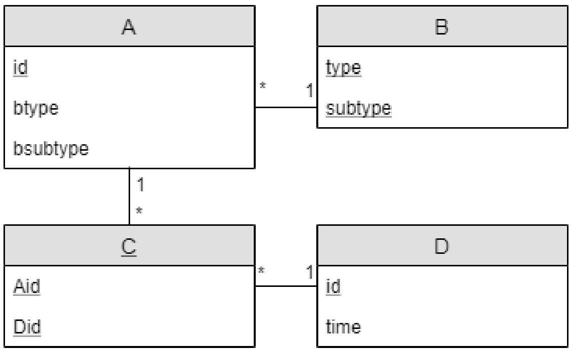

Figure 2.2: Example ER model

A		
id	**btype**	**bsubtype**
1	type 1	subtype 1
2	type 2	subtype 1

B	
type	**subtype**
type 1	subtype 1
type 2	subtype 1
type 2	subtype 2

C	
Aid	**Did**
1	1
1	2
2	1

D	
id	**time**
1	1
2	2
3	1

Table 2.2: Example data following the ER model shown in Figure 2.2

2.3 Graph databases

A graph database is a database that stores and represent data using graph structures. A graph is a collection of nodes and relationships. There are multiple ways that graphs can be represented, such as hyper graphs, triples or labeled property graphs. We will be using labeled property graphs. A labeled property graph $G = (N, R)$ consists of nodes N and relationship R. Each node $n \in N$ has a label $n.label$. Each relationship $r \in R$ has a type $r.type$. Both nodes and relationships can have any number of properties. For the sake of consistency with other concepts in this thesis, we will refer to these properties as 'attributes' instead. attributes can be accessed in two ways. Say n has an attribute called p. We will use the notation $n[p]$ to access attribute p of node n. An attribute of a relationship can be accessed in the same way.

2.3.1 Neo4J

In this thesis we will work with a graph database system called Neo4J [1], which uses property graphs to store data.

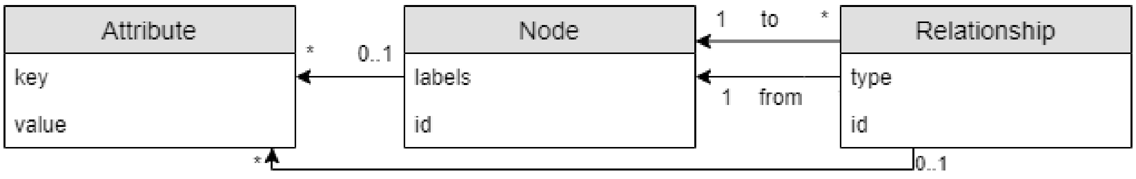

Figure 2.3: Property graph meta model

Data stored in a Neo4j graph database can be described by the metamodel shown in Figure 2.3. Such data consists of nodes, relationships and attributes. Nodes can have multiple labels to indicate their role(s) in the graph and can have any number of attributes (key-value pairs). Relationships directionally connect two nodes to indicate that they are somehow related. They can have a single type, to indicate in what way the two nodes are related and can also hold any number of attributes. Nodes and relationships are uniquely identifiable by their node ids and relationship ids respectively.

2.3.2 Patterns and pattern matching using Cypher

To query the data, Neo4J uses a declarative query language called Cypher [2]. Cypher makes heavy use of patterns. Using patterns you describe the shape of the data you are looking for. Cypher will then 'match' the part of the graph that conforms to the provided pattern.

The simplest pattern that can be defined, is that of a node, which is defined as `(a)`. The *a* is a variable which we can later use to refer to nodes that match this pattern. We can also define patterns to describe multiple nodes and relationships between them. For instance, the pattern `(a)-->(b)` matches nodes that have a relationship between them. This pattern has two variables *a* and *b*. This can be extended arbitrarily, e.g.`(a)<--(b)--(c)`. We call such a series of connected nodes and relationships a path.

In addition to specifying the shape of the pattern, you can also introduce some constraints to patterns. Take for instance the pattern `(a:Person {gender: "male" , age: 35})`. Here we match all nodes with a label 'Person' and attributes 'gender' equal to 'male' and 'age' equals to 35.

Cypher also allows you to assign variables to relationships, for example `(a)-[b]->()`. Here we assign the variable *b* to the relationships that match this pattern. Note that you don't have to assign a variable to each pattern component and can simply leave the component empty.

Like with nodes, constraints can also be specified for relationships. Consider the pattern `()-[a:Friend {blocked: false}]-()`. Here we specify that we want to find all relationships of type 'Friend', with the attribute 'blocked' equal to 'false'.

Finally, there exists a construct to describe a sequence of relationships, which we call variable-length pattern matching. Consider the pattern `(a)-[*2]->(b)`. The *2 indicates that we are looking for nodes that are two relationships away from each other. This pattern is equal to the pattern `(a)-->()-->(b)`. Now consider the pattern `(a)-[*2..4]->(b)`. Here we look for nodes that are two to four relationships away from each other. Finally consider the patterns `(a)-[*2..]->(b)`, `(a)-[*..2]->(b)` and `(a)-[*]->(b)`. Here we are looking for nodes at least two, at most two and any number of relationships away from each other respectively.

Note that variable length pattern matching can, depending on the size and type of graph, be an expensive operation, especially when the number of relationship steps is variable, like with the last three patterns. This is because nodes might have multiple relationships that satisfy the relationship criteria of the pattern, which will all have to be 'explored' by the database engine. The target nodes of those relationships can in turn have multiple of such relationships, etc. This can lead to a complicated search space, which means it takes a long time for the database engine to answer such a query precisely.

2.4 R2PG-DM

R2PG-DM [9, 10], is a tool that is able to take any SQL database and transform it into a graph representation of that database. In this graph representation, every row of the input database is modelled as a node in the graph, with relationships between nodes where foreign key relations existed in the input database.

There exist several graph database engines, each with their own method of importing data or querying. R2PG-DM aims to output a generic file format to represent graphs, such that this file format can easily be transformed to meet the graph database engine's specific way to import the data. Section 2.4 explains this, and how we deal with it, in more detail

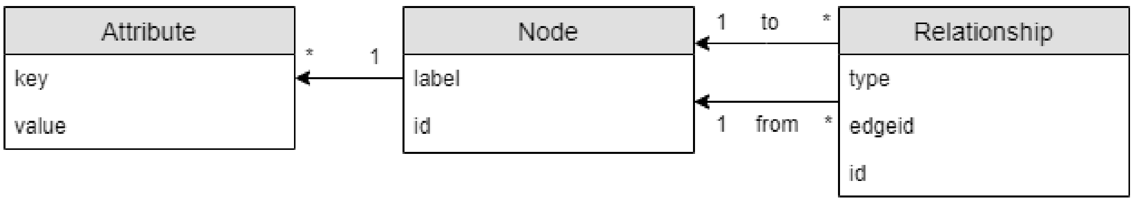

Figure 2.4: R2PG-DM meta model

Figure 2.4 shows the meta model of data R2PG-DM produces. This model is only slightly more restrictive than the model that describes Neo4J's property graphs, shown in Figure 2.3. It has all the same elements, but nodes can only have a single label; and relationships do not have attributes and have an extra 'edgeid' attribute. This extra attribute uniquely defines all relationships, but Neo4J calculates its own relationship identifiers, which are stored in the 'id' attribute of relationships, so we can ignore this new attribute when working with Neo4J.

Even though the output has a pretty non-restrictive meta model, we can easily work with it, due to the way R2PG-DM uses it. Consider the example data set shown in Table 2.3, which follows the ER diagram shown in Figure 2.5. This data set contains four tables. The following foreign key relations are present:

- *A.btype* and *A.bsubtype* together reference *B.type* and *B.subtype*
- *C.Aid* references *A.id*
- *C.Did* references *D.id*

Figure 2.6 shows the graph that R2PG-DM would produce using the example data set as input. Every row of our example data set has been transformed into a node. We call this row the source row of that node. The node's type corresponds to the name of its source row's table. Each row's columns are present as node attributes and all nodes are uniquely identified with a node identifier. The foreign key relations between the rows have been transformed into relationships between the nodes representing those rows. Note that the direction of these relationships is consistent with the direction of the foreign keys they represent. The type of each of such relationships is obtained by concatenating the label of the source and target node, separated by a hyphen.

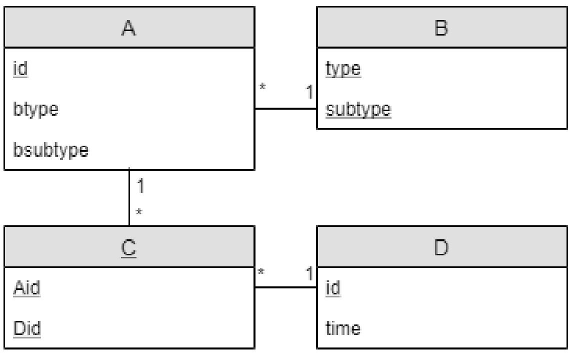

Figure 2.5: Example ER model

A		
id	**btype**	**bsubtype**
1	type 1	subtype 1
2	type 2	subtype 1

B	
type	**subtype**
type 1	subtype 1
type 2	subtype 1
type 2	subtype 2

C	
Aid	**Did**
1	1
1	2
2	1

D	
id	**time**
1	1
2	2
3	1

Table 2.3: Example data following the ER model shown in Figure 2.5

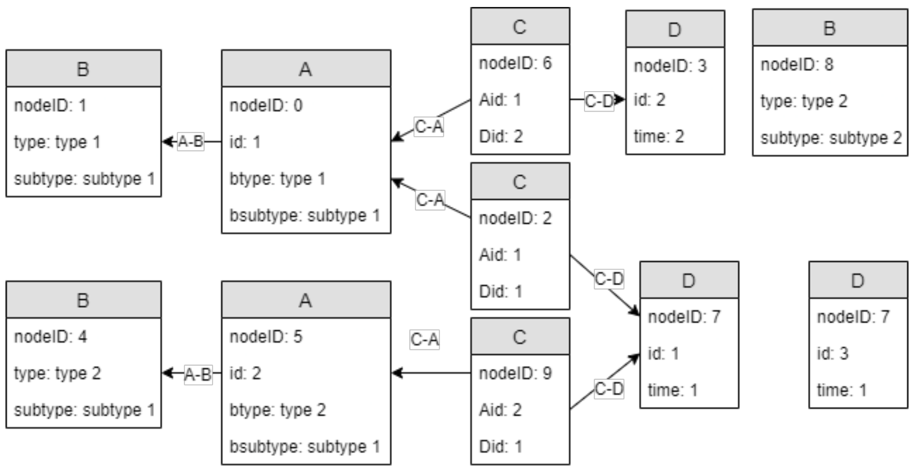

Figure 2.6: Property graph as a result of R2PG-DM transformation of data shown in Table 2.3

2.5 Business process intelligence challenge 14

The business process intelligence challenge 14 (BPI 14) [12] was a challenge held in 2014 where participants were asked to analyze data provided by the Rabobank Group ICT. The provided data set contains record details from an ITIL Service Management tool called HP Service Manager and consists of four *.csv* files, which are described by Figure 2.7. Here you can see a schema for four tables, one for each *.csv* file and three documented foreign key relations. We will explain the BPI 14 data set in more detail in this Section and will use it as a running example during this thesis.

2.5.1 Data normalization

The data set as provided by BPI 14 is quite denormalized and the change table cannot be directly related to any other table. This denormalization is most visible for columns that describe configuration items (CI) and service components, as these columns are present in the Incident, Interaction and Change, while they could be in two separate tables and referenced via a foreign key instead. This means it does not meet the requirements we will set for relational data schemas that can be used for the transformation described by this thesis. These requirements will be discussed in Section 3.1. For this reason we will normalize the data set[1]. The normalized schema can be seen in Figure 2.8.

These new tables have been derived as follows:

- **Service_Component**: The Incident, Interaction and Change tables all contain columns describing service components. Some of these columns end in 'aff', while others end in 'CBy'. There are two different suffixes, because the Incident table has columns describing two different service components, so this way those column names can be differentiated. A new table is created with a single 'ID' column containing all distinct values of the service component columns of the aforementioned tables.

[1] [7] shows how artifacts can be discoverd and extracted from denormalized data.

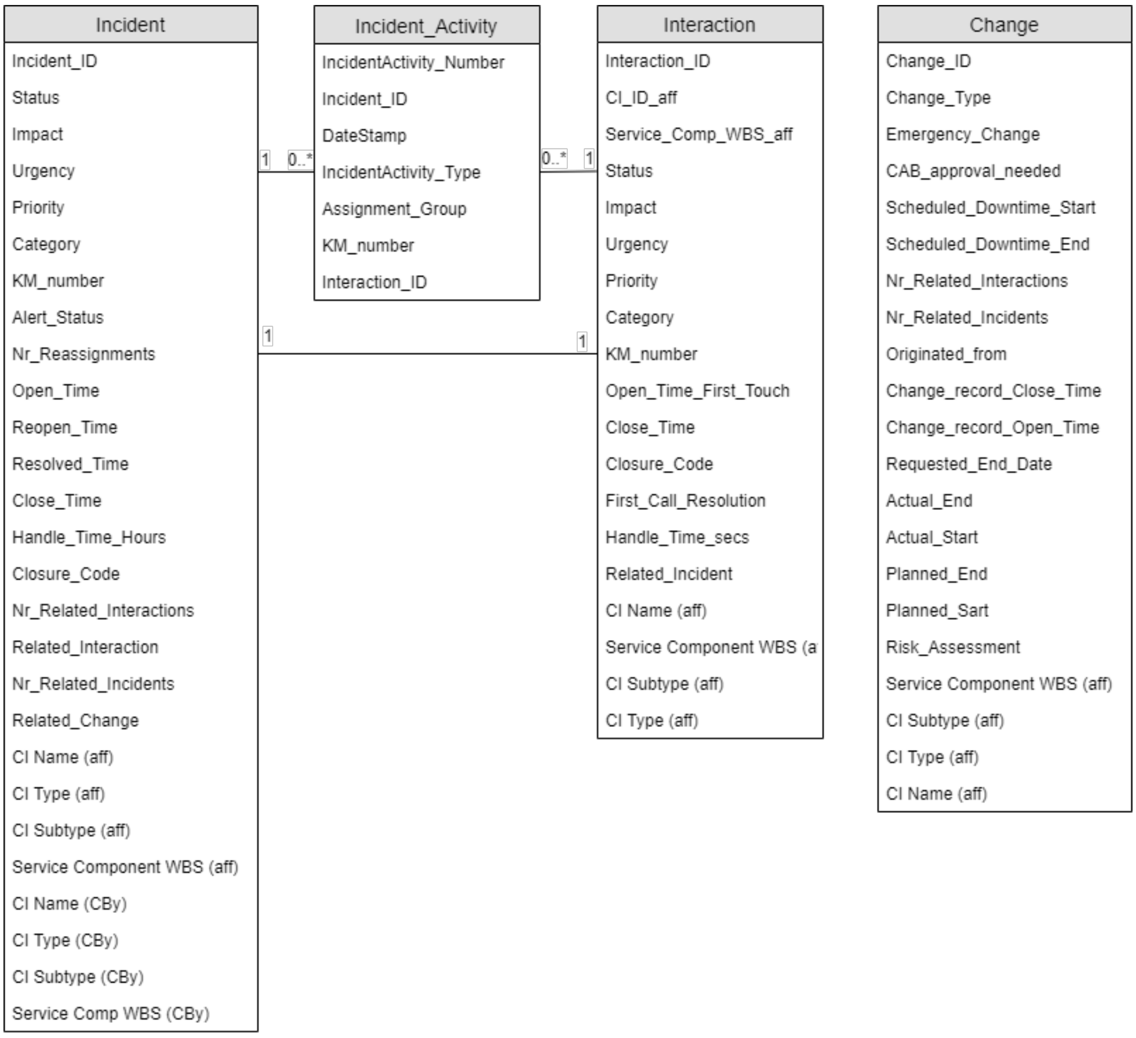

Figure 2.7: ER diagram of the BPI 14 data set

- **Configuration_Item**: Configuration items are stored in the same tables of the original database as service components. Columns describing configuration items contain 'CI' in their name. The 'aff' and 'CBy' suffixes serve the same purpose as they did for service components. We create a 'Configuration_Item' table with the aforementioned columns, but without their suffixes. In addition to these columns, The new Configuration_Item table has an 'ID' column, which is its primary key and a 'Service_Component' column, which references the Service_Component table.

  Each configuration item can be distinguished by a name, a type, a subtype and a service component, therefore each distinct combination of these values is added as a tuple to the new Configuration_Item table.

- **Assignment_Group**: The new Assignment_Group table has a single column 'ID' which contains values from the 'Assignment_Group' column of the Incident_Activity table. Many Incident_Activity rows share the same assignment group, which is why we decided to collect these assignment groups in a new table.

- **Knowledge_Document**: The Incident_Activity, Interaction and Incident tables all have a 'KM_number' column. These columns all refer to the same group of (knowledge document)

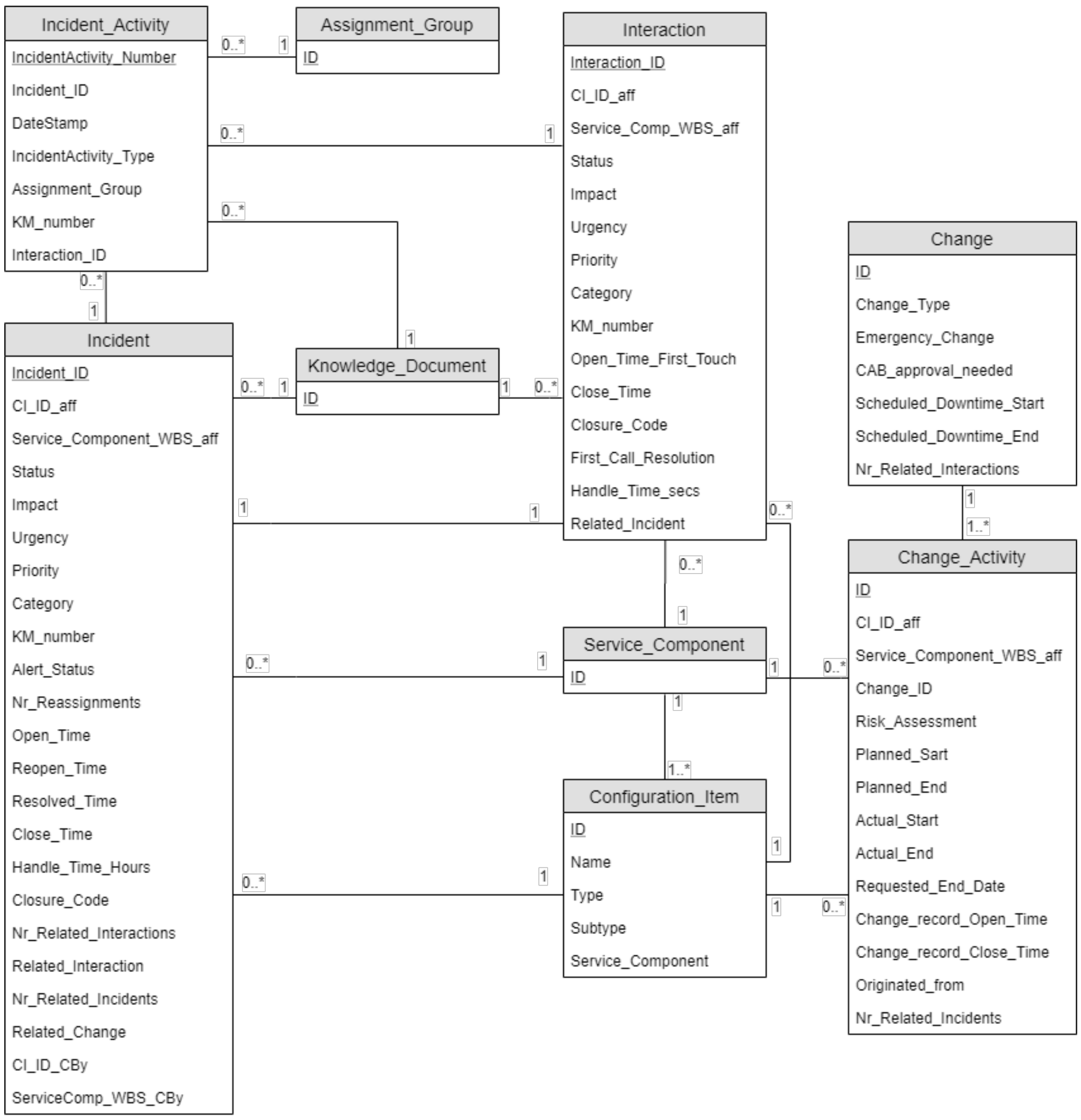

Figure 2.8: ER diagram of a normalized BPI 14 data set

entities. The new table has a single 'ID' column containing values from these other columns.

- **Incident**: The 'Incident' table remains largely unchanged, but we make the following changes. We replace the two configuration item name, type and subtype columns with two columns 'CI_ID_aff' and 'CI_ID_CBy', which are both foreign keys to the 'ID' column of the Configuration_Item table. Furthermore both service component columns are foreign keys to the 'ID' column of the Service_Component table. Finally the 'KM_number' is a foreign key to the 'ID' column of the Knowledge_Document table.

- **Interaction**: The Interaction table undergoes the same changes as the Incident table. The only difference is that the Interaction table only has columns for one service component and configuration item, instead of two.

- **Incident_Activity**: The only change to this table is that it gets several foreign keys.

- ‘Incident_ID’ references the ‘Incident_ID’ column of the Incident table.
- ‘Interaction_ID’ references the ‘Interaction_ID’ column of the Interaction table.
- ‘Assignment_Group’ references the ‘ID’ column of the Assignment_Group table.
- ‘KM_number’ references the ‘ID’ column of the Knowledge_Document table.

- **Change and Change_Activity**: The Change table is split into these two tables, where each table gets a subset of the original Change table’s columns. This is because the ‘Change_ID’ column does not uniquely identify each tuple in the Change table, despite there being a *Change_ID* column. Therefore the Change table keeps all columns that stay consistent among all rows with the same ‘Change_ID’. All other columns are added to a new Change_Activity table, along with the ‘Change_ID’ column and a new ‘ID’ column, which is the primary key of this table. The ‘Change_ID’ column of the Change table is renamed to ‘ID’.

  Just like with the Interaction table, the configuration item and service component columns are replaced with columns that reference the Configuration_Item and Service_Component tables. Finally the ‘Change_ID’ column is a foreign key to the ‘ID’ column of the Change table.

2.6 Business process intelligence challenge 17

The business process intelligence challenge 17 (BPI 17[13]) was a challenge held in 2017 where participants were asked to analyze an event log provided by a Dutch financial institute. This event log describes the application process for a personal loan or overdraft within a global financing organization. The event log consists out of a single *.xes* file, which we converted to a *.csv* file using a ProM plugin. Figure 2.9 shows an ER diagram of the data in this *.csv* file. Note that this ER diagram shows no primary key, as the data contains duplicate rows. Each row of this *.csv* file corresponds to an event. Each event belongs to either an application, offer or workflow, which can be considered as entities. each event has a column ‘EventOrigin’, which allows us to identify whether the event belongs to an application, offer or workflow.

2.6.1 Data normalization

Like BPI 14, BPI 17 is too denormalized for our needs. We identified 7 entity types, which are all situated in the provided event log. To normalize this data set, we extracted all these 7 entity types into 7 tables, which we stored in an SQL database. An ER diagram of our normalized BPI 17 data set is shown in Figure 2.10. The various tables of this ER diagram were constructed as follows:

- **resources**: The ‘resource’ table consists of a single column, which contains the unique values of Figure 2.9’s ‘org:resource’ column. We renamed this ‘org:resource’ column to ‘resource’ in this, and all upcoming tables.

- **applications**: The ‘application’ table gets all its columns from Figure 2.9. Assume this is true for the remaining tables below as well. we renamed the ‘case’ column to ‘ApplicationID’, which also serves as the ‘application’ table’s primary key.

  To determine which rows the ‘application’ table should contain, we take all rows of the BPI 17 data set and keep only one row per distinct ‘case’ value and add these rows to the ‘application’ table (only the columns listed for the ‘application’ table are taken). We can do this because every two rows with the same ‘case’ value, also have the same values in the other columns we require for the ‘application’ table

- **application_events**: we renamed the ‘case’ column to ‘ApplicationID’ and added a new column ‘ID’ to serve as this table’s primary key. The ‘ApplicationID’ column also serves as a foreign key to the ‘applications’ table’s primary key. Furthermore, the ‘resource’ column

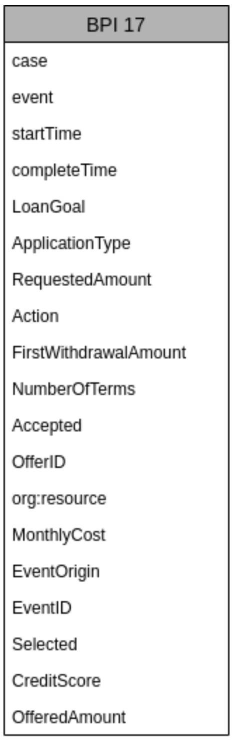

Figure 2.9: ER diagram of the BPI 17 data set

serves as a foreign key to the 'resources' table's primary key and the 'OfferID' column serves as a foreign key to the 'offers' table's primary key. 'OfferID' is added as a foreign key, as many 'application_events' rows list an 'OfferID'

All of BPI 17's rows contains a single event, but not every event belongs to an application. The 'application_events' table contains all BPI 17's rows with 'EventOrigin' equal to 'Application'

- **offers**: We renamed the 'case' column to 'ApplicationID' and made the 'OfferID' column this table's primary key. To determine which rows the 'offers' table should contain, we do the same as for the 'applications' table, but for the 'OfferID' column of Figure 2.9.

- **offer_events**: We added a new column 'ID' to serve as this table's primary key. This table has two foreign key columns. 'OfferID' refers to the 'offers' table's primary key and 'resource' to the primary key of the 'resources' table. The 'offer_events' table contains all BPI 17's rows with 'EventOrigin' equal to 'Offer'

- **workflows**: Workflows exist in a one-to-one relation with applications and can thus be identified with the same identifier as applications. Therefore we renamed the 'case' column to 'WorkflowID'. To determine which rows the 'offers' table should contain, we do the same as for the 'applications' table.

- **workflow_events**: we renamed the 'case' column to 'WorkflowID' and added a new column 'ID' to serve as this table's primary key. The 'WorkflowID' column is a foreign key to the 'applications' table's primary key. The 'OfferID' and 'resource' columns are foreign keys to the 'offers' and 'resource' tables' primary keys respectively. 'OfferID' is added as a foreign key, as many 'workflow_events' rows list an 'OfferID'. The 'workflow_events' table contains all BPI 17's rows with 'EventOrigin' equal to 'Workflow'

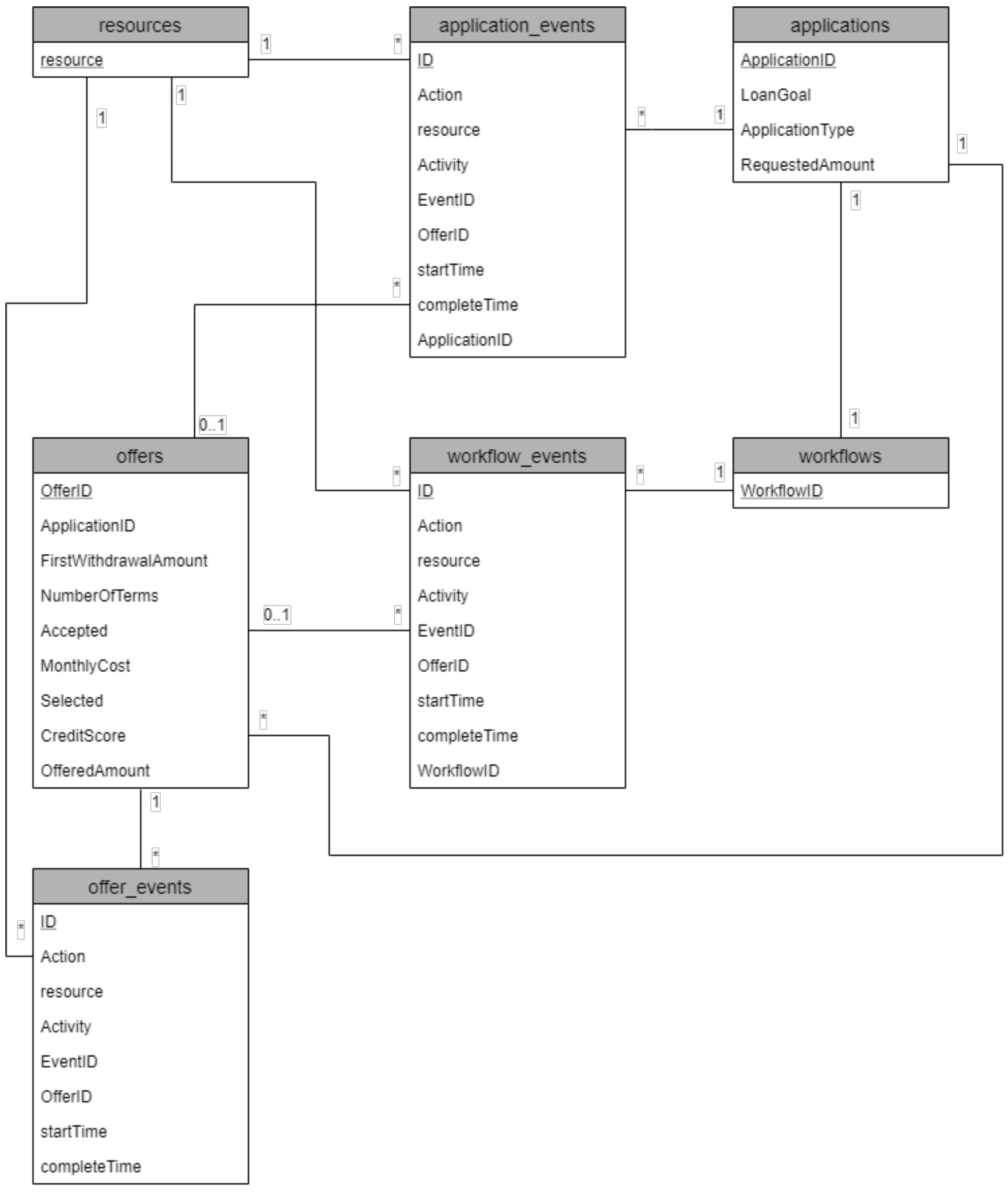

Figure 2.10: ER diagram of a normalized BPI 17 data set

Chapter 3

Data Models

In this chapter we discuss the various data models that are used during the transformation from relational data to event graph data. We start with discussing to what requirements input data needs to conform so that it can be transformed into graph data capturing both the input data's structural and temporal relations in Section 3.1.

Next, we will discuss requirements of a data model to represent normalized event data in a graph database in Section 3.2. Before we propose our data model for this purpose in Section 3.2.6, we discuss model requirements for events and entities in a labeled property graph in Section 3.2.1 and two other relevant data models. First we recall the data model in which R2PG-DM outputs data in Section 3.2.2 and then we discuss an existing data model for storing event data in labeled property graphs and its shortcomings in Sections 3.2.3 and 3.2.4. We will use this existing data model as the basis for the data model we propose and present several design options that improve the existing data model in Section 3.2.5. After presenting these options, we will decide for a design option in Section 3.2.6 and highlight the differences between our proposed data model for representing event data in labeled property graphs and the existing one in Section 3.2.7.

Finally we discuss the concept of compound entities and how 'directly follows' relations can be represented for compound entities in Section 3.3.

3.1 Relational data: required schema and data

An entity is anything about which information can be stored, for example a person, a transaction, an activity, or an information object that a process works on such as a credit application. In this section we discuss how we assume entities to be represented in the relational data. We need to make this assumption because we later want to represent entities as nodes and we want our transformation to preserve these entities.

In order for a relational database to be used as input to our transformation, it will have to conform to these assumptions:

1. The schema needs to be normalized in a way such that each table that contains entities, contains exactly one entity per row. Each of these tables must also only contain entities of only a single type.

   Take the normalized BPI 14 data set as an example. Each row in the Incident table describes one incident entity. Each row in the Incident_Activity describes a change to the incident, which we would call an Incident_Activity entity.

   We need these entities to be represented like this, because they are needed to generalize case identifiers of event logs. This way we can assign these generalized case identifiers to events in a structured way. Otherwise, each entity type could be spread out over multiple rows of (potentially) different tables, making them very difficult to structurally access.

2. There needs to be at least one table with one or more timestamp columns, but there can be more relations with timestamp columns. These are needed to create events, as events need some attribute on which they can be ordered.

3. While not a hard requirement, it is preferable that each table which does not have a timestamp column, gets referenced by a table that does, or gets referenced by a table that in turn gets referenced by a table that does, etc. Alternatively, the table can be left out of the transformation entirely. Using such a table in the transformation results in entities that do not relate to any events. While this does not technically violate the data model we propose in Section 3.2.7, it also adds no value to the resulting event graph for this reason.

4. Each timestamp must either exist in the same row as the entity to which it belongs, or exist in a row that references (via a chain of one or more foreign key relations) the entity/entities to which it belongs. We need this to hold, because entities form the context for multiple events like case identifiers do in an event log and allow us to consider multiple events together. If this requirement does not hold, we can not relate events with such timestamps with each other.

5. The primary key of each table containing entities needs to consist of exactly one column. This is a limitation of our transformation.

3.2 Data model for event data in labeled property graphs

The event graph data model is what will be used to store the transformed relational data described by Section 3.1. In order for process mining algorithms to use this data, the algorithm needs to be able to assume a certain data model. This section will thus discuss the requirements of a data model to represent normalized event data in a graph database.

3.2.1 Model requirements for events and entities in a labeled property graph

The data model needs to be able to represent a couple of concepts. First, it needs to store entities and their attributes, which we discussed in Section 2.4. Next, we need to be able to represent events. For each event we need to at least know their start time, as well as a name to describe the activity performed in the event. If an event has an end time, in addition to a start time, that end time needs to be modelled as well. Each event can involve one or more entities, so we need to be able to find which entities are involved in an event. In other words, the graph needs to represent the one-to-one, one-to-many and many-to-many relations between events and entities.

Furthermore, we want the model to represent a temporal relation between events, namely the 'directly follows' relation, which is often used in process mining algorithms and can be used to derive many other temporal relationships, like the 'eventually follows' relationship. We want to model these 'directly follows' relations on the level of entities. In other words, we want to model for each entity what the temporal ordering of the start time of the events related to that entity is.

All of the above needs to be modelled in such a way that it is efficient to query and store, and easy to query and analyse. Querying for the temporal relations needs to be efficient in particular, as those relations are almost always needed by process mining algorithms.

3.2.2 labeled property graph schema as created by R2PG-DM

The schema produced by R2PG-DM as explained by Section 2.4 is good, but doesn't know about entities and events. However it is a good data model to base our transformation on.

3.2.3 Esser's data model for event data in labeled property graphs

In previous work [5], Esser proposed a data model to store event data in graph form. His data model features three node labels; Log, Entity and Event and four relationship labels; E_EN, DF, L_E and HOW. Figure 3.1 shows an ER diagram of this model. We will refer to this model as the original data model. Note that named edges refer to relationships in the graph. Boxes with a grey header refer to nodes and boxes with a white header refer to relationships in the graph that can have attributes.

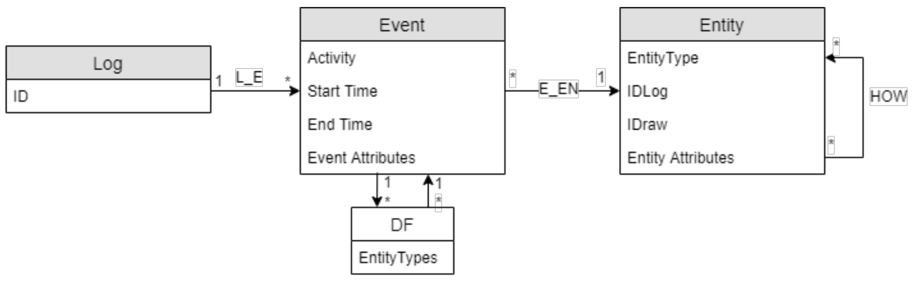

Figure 3.1: ER diagram describing the existing data model

Nodes

The Log node represents a log of a single process. One graph can contain multiple Log nodes from possibly different processes. Each Log node has one *ID* attribute that indicates the log's name/identifier.

Each Entity node represent an entity in the data. Each Entity node has several default attributes;

- *EntityType* specifies the type of the Entity node, e.g. 'Person'.
- *IDLog* is a unique Identifier to identify the entity across all logs in the graph.
- *IDraw* is the id used to identify this entity in the original data.

Entity nodes replace the notion of case identifiers, as in this data model, events can relate to multiple cases, which means one case identifier is no longer sufficient. The Entity node also contains entity attributes that describe that entity. An entity attribute is an attribute of an entity that does not change during the lifespan of that entity. They are comparable to *case attributes*, which are attributes that never change during the case's lifespan.

The Event node represents events in the data. Each event in the data is represented by one Event node. Every Event node has at least an *Activity*, *Start Time* and *End Time* attribute, indicating the event type, start and end time respectively. Furthermore, an event can have any number of event attributes.

Relationships

The E_EN (Event to Entity) relationship connects events to entities. Each event is connected to one or more entities and each entity can have incoming relationships from zero or more events. If an event is connected to an entity using an E_EN relationship, it means that the entity was involved in the event. In other words, that entity is a kind of case identifier of that event.

The DF (Directly Follows) relationship connects events to other events. The source and target Event nodes must be related to the same Entity nodes via an E_EN relationship. This corresponds to events being related to the same case identifier in an event log. It means that for some entity, related to both the source and the target event of the DF relationship, the events directly follow each other. The target event of such a DF relationship always occurs after its source event. In other words, for each DF relationship, there is no Event node with both a start time later than the start time of the DF relationship's source node and earlier than the start time of the DF relationship's target node, where all three of these Event nodes are connected to the same Entity node. Each DF relationship contains an attribute EntityTypes, which holds a list of entity types for which this DF relationship holds.

The HOW (Handover Of Work) relationship connects entities to other entities. An entity can be connected to zero or more entities of the same type via these relationships. If two entities are connected via a HOW relationship, it means that there exists a DF relationship between two events, where the source entity is connected to the source event and the target entity is connected to the target event, i.e., The source entity 'handed over' work to the target entity.

The L_E (Log to Event) relationship connects Log nodes to Event nodes. Every Event node is connected to one log node, and every Log node is connected to one or more Event nodes. This relation indicates that an event is part of the connected log. A Log node is never reachable from another Log node, i.e. The sub-graph of any Log node, which consists of the nodes reachable from that Log node via the above relationships, is disconnected from the sub-graph of another Log node.

3.2.4 Shortcomings of existing data model

We found that the data model discussed above has some shortcoming that we'd like to address.

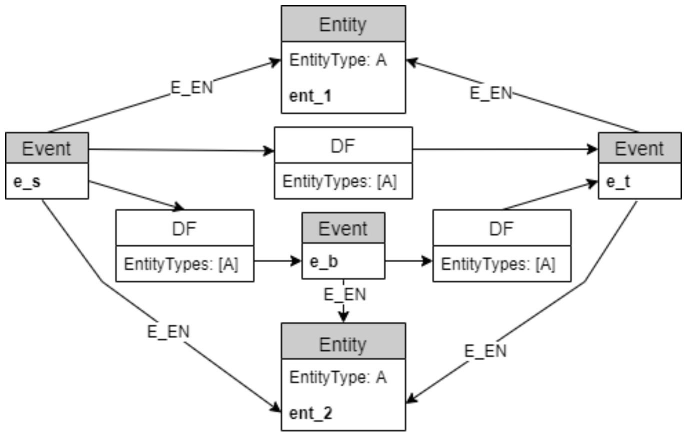

Figure 3.2: Instance of original event data model, showing an unclear DF path

The first issue is related to DF relationships. Consider the labeled property graph (i.e, an instance of the schema of the original data model) in Figure 3.2. Let e_s, e_b, e_t be Event nodes and ent_1, ent_2 Entity nodes with the same entity type. The source and target Event nodes, e_s, e_t, of a DF relationships both have an E_EN relationship to both Entity nodes. For ent_1 it holds that e_t directly follows e_s, but ent_2 is connected to another Event node e_b, such that e_b directly follows e_s and e_t directly follows e_b. There is no problem with the DF path (the path of DF relationships, such that all source and target Event nodes of those DF relationships are connected to that Entity node via an E_EN relationship) of ent_1. However, if we look at the DF path of ent_2, we will find that both e_t and e_b directly follow e_s, with no way to determine whether e_t or e_b truly directly follows e_s without looking further along the DF graph, as both relationships simply state that they relate to an entity with ent_2's entity type. This makes it difficult to understand and use the data.

Furthermore, it is quite expensive to perform certain queries, like calculating whether some Event node eventually follows another Event node, due to the way DF relationships are structured. This is because one event can have multiple incoming and outgoing DF relationships, possibly of the same type. This means that the database engine has to 'visit' multiple paths when answering a query like this, like we described in Section 2.3.2.

In order to resolve these issues, we need the following properties to hold.

1. For each DF relationship, we can find to which entity/entities it applies, by only looking at that relationship, its source and target nodes and the entities related to those nodes via a E_EN relationship.

2. There is no Event node with two or more outgoing DF relationships with the same type.

3.2.5 Design options for improving the data model

There are several ways to go about improving the original data model such that the above properties hold. We will focus on Event nodes, DF relationships and Entity nodes. Figure 3.3 shows a possible instance of an event graph with 3 Entity nodes and 4 Event nodes in Esser's data model, where we focus on the main Event node with *ID: 0*. The main Event node has an incoming DF relationship, related to three entity types and two outgoing DF relationships, together related to the same three entity types. For clarity, we have drawn the E_EN relations between events, other than the main event, and entities in a lighter shade of grey. Also, the *ID* attribute is only added to the figures to easily differentiate between different events and entities.

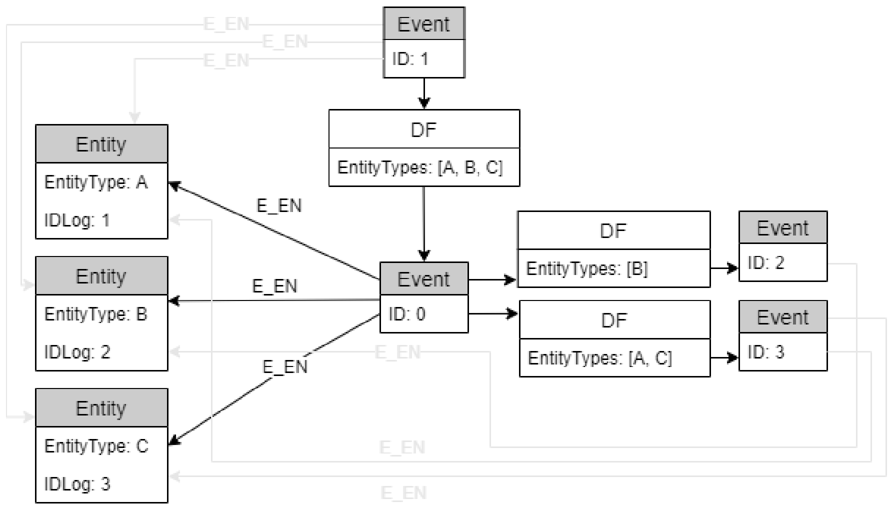

Figure 3.3: Instance of Event node structure using the original data model

Design option 1

Consider the data model shown in Figure 3.4. Figure 3.5 shows an instance of this data model.

1. Property 1 is addressed by adding a list of identifiers on each DF relationship, which contains all the *IDLog* identifiers of all entities related to that relationship. We store this list on the *EntityIDs* attribute of the DF relationship. This allows you to immediately identify if a DF relationship is related to any entity.

2. Property 2 is addressed somewhat by having a specific DF relationship type for each entity type. This way, DF relationships with a different type don't need to be explored when calculating eventually follows relations between events, as we discussed in Section 3.2.4. Instead there would only be one path of DF relationships to follow, unless the graph contains an event that is connected to multiple entities of the same type. In that case, property 2 does not hold for that graph.

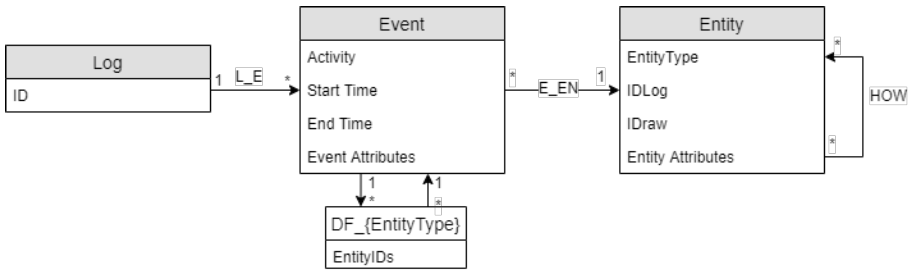

Figure 3.4: ER model of design option 1

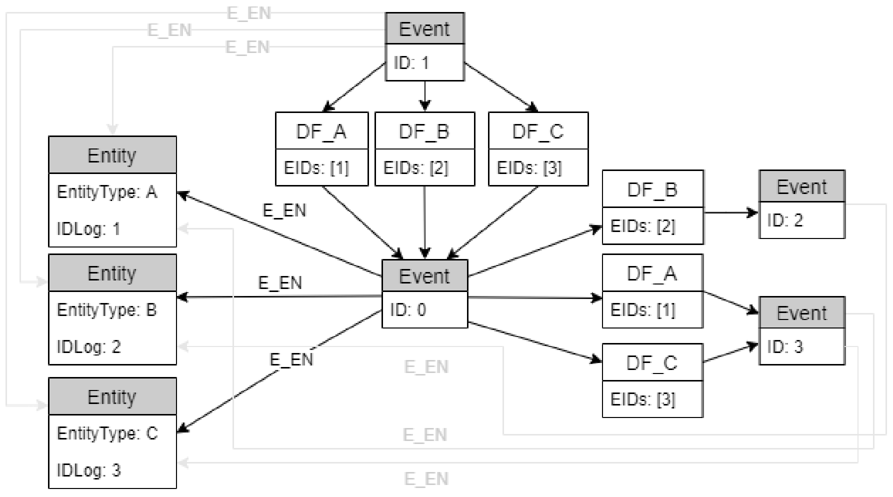

Figure 3.5: Instance of Event node structure using design option 1

Design option 2

Now consider the data model shown in Figure 3.6. Figure 3.7 shows an instance of this data model. We create multiple LEvent (Local Event) nodes for one event, one for each entity related to that event. Together they form one event. Each of these LEvent nodes is connected to one entity, such that all related entities are connected to one of the LEvent nodes. Each LEvent node keeps the minimally required properties, which are the same over each of these LEvent nodes. We need to be able to find other entities related to this event, which is why we need to have relationships between the LEvent nodes that originate from the same event.

Here we choose to make the LEvent nodes fully connected using E_EN relationships. Note that the instance of Figure 3.7 only shows this for the main event, but all events with ID 1 are fully connected, as are the events with ID 2 and 3. To summarize:

1. Property 1 is addressed, since every DF relationship has a source and target node that exclusively relate to one Entity node via an E_EN relationship
2. Property 2 is addressed, since we create several LEvent nodes for each event, one for each related entity, which means that each LEvent node has at most one incoming and one outgoing DF relationship.

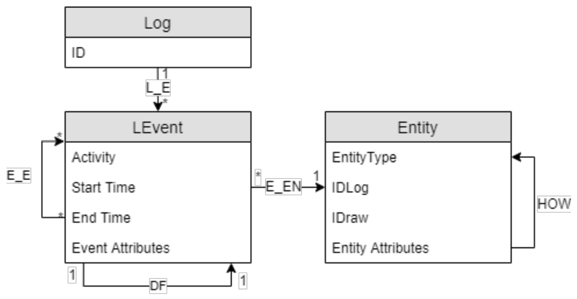

Figure 3.6: ER model of the event graph data model using design option 2

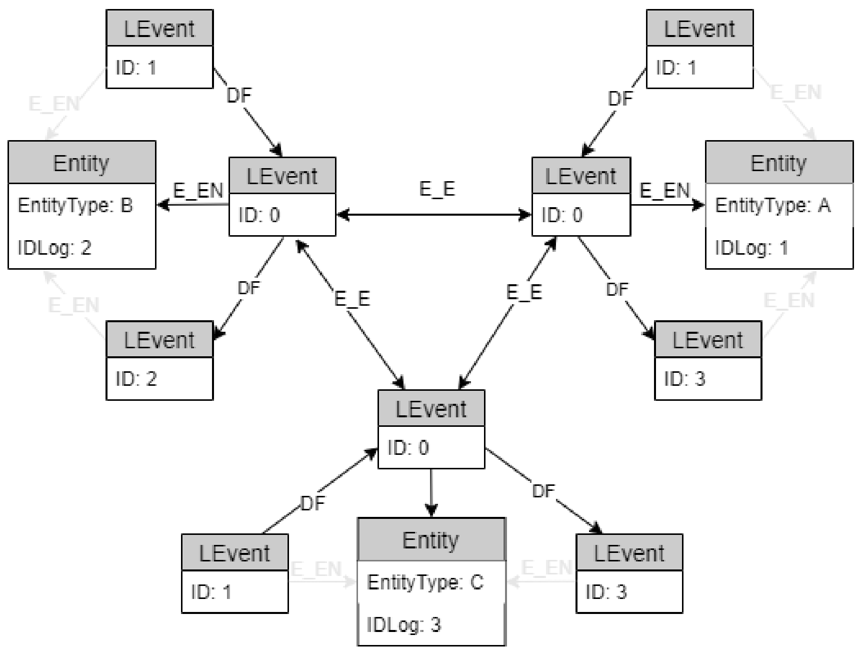

Figure 3.7: Instance of LEvent node structure using design option 2

Design option 3

Finally consider the data model shown in Figure 3.8. Figure 3.9 shows an instance of this data model. As is clear from this figure, this design option is nearly identical to the second design option. It addresses both properties in the same manner, but we use a different way to connect events nodes that represent the same event.

One Event node is introduced per event. All LEvent nodes based on the same event have a LE_E relationship to the Event node that's also based on that event. Like with design option 2, note that Figure 3.9 only shows this Event node construct for the main LEvent node '0', but it also exists for the others. The Event node only serves to group LEvent nodes that represent the same event, such that from each LEvent node, we can find all other LEvent nodes related to the same event by visiting the related Event node. To summarize how each of the desired properties is addressed, we repeat what was stated for design option 2:

1. Property 1 is addressed, since every DF relationship has a source and target node that exclusively relate to one Entity node via an E_EN relationship
2. Property 2 is addressed, since we create several LEvent nodes for each event, one for each related entity, which means that each LEvent node has at most one incoming and one outgoing DF relationship.

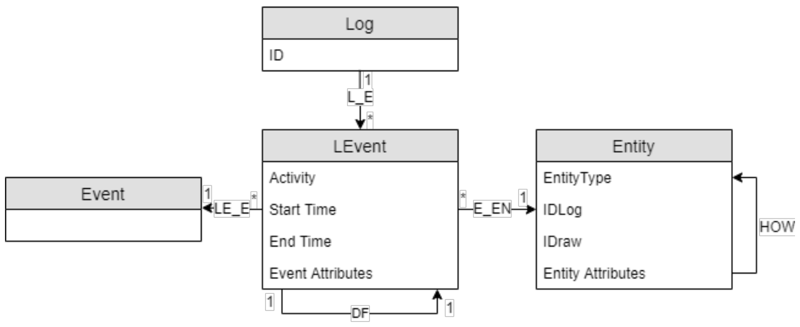

Figure 3.8: ER model of the event graph data model using design option 3

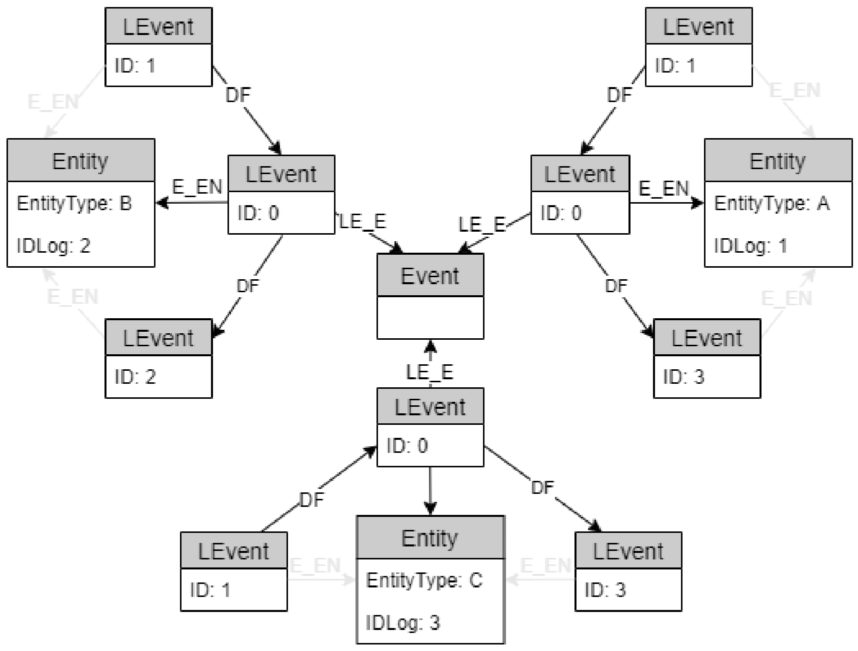

Figure 3.9: Instance of LEvent node structure using design option 3

3.2.6 Deciding for a design option

Now that we have discussed some design options we can make, we need to decide on one of them.

Design option 1 addresses the first of our desired properties, but it can not guarantee that the second property holds. Furthermore, having multiple types of DF relationships, makes writing queries difficult, as now you need knowledge about what types are in the data in order to specify DF relationships.

Design option 2 address both of the desired properties. We also gain the property that each LEvent node can have at most one incoming and one outgoing DF relationship, as each LEvent node is connected to only one entity and every entity can only have one path of DF relationships in which each connected event is visited once on that path. This also means we don't need to keep the entity types and ids on the DF relationships.

A downside of design option 2 is that it is harder to formulate some queries. For instance, to retrieve all entities related to an event in the original model, we would just query for all nodes related to that event via a E_EN relationships, but using option 2, we need to query for all nodes that can be reached by traversing an E_EN relationship and then an E_EN relationship. Another downside is that it is more expensive to store the data, since we need to store each event using multiple nodes and relationships, instead of just one node. More specifically, we need $n(n-1)$ relationships per event, where n is the number of entities related to the event. This is shown in Figure 3.10.

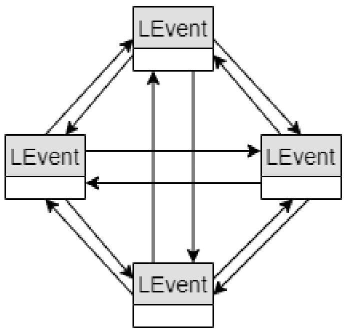

Figure 3.10: Example of design option 2, showing the number of relationships required to connect all LEvent nodes of one event

Design option 3 has the same benefits and downsides as the second design option, but reduces the number of required relationships between LEvent nodes, as here we need just n relationships to connect the LEvent nodes with each other, as shown in Figure 3.11.

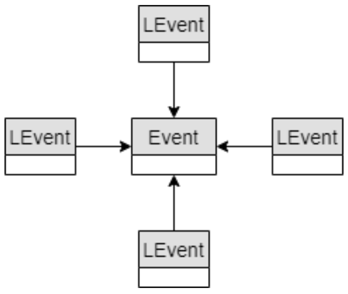

Figure 3.11: Example of design option 3, showing the number of relationships required to connect all LEvent nodes of one event

Since the design option 1 does not always address all desired properties, and design option 3 is basically an improvement over design option 2, we will implement design option 3 into the design of the final labeled property graph event data model.

3.2.7 Final data model for event data in labeled property graphs

Now that we have decided how to address the issues we described in Section 3.2.4, we will highlight the differences between the original data model and the data model we will be using. The ER diagram of this model is shown in Figure 3.11.

We renamed the Event node to LEvent node and introduced the Event node. For each event in the data set, there exists one Event node and one or more LEvent nodes in the graph, one per entity related to that event. The Event node does not have any attributes. Each of the LEvent nodes has the minimally required event attributes, is connected to the event's Event node via an LE_E (LEvent to Event) relationship and is connected to exactly one Entity node via an E_EN relationship.

Each LEvent node can now have at most one incoming and one outgoing DF relationship. Two LEvent nodes connected by a DF relationship must both have an E_EN relationship to the same Entity node. DF relationships are no longer required to store a list of entity types, but optionally, a single entity type and/or entity id can be stored on a DF relationship.

3.3 Representing 'directly follows' relationships for compound entities

In both the original and new data model, DF relationships represent temporal relations between events based on one entity. Consider the scenario sketched in Figures 3.12 and 3.14, which show an instance of the original and new data model respectively. Both entities A and B are related to one shared event (e1) and one non-shared event (e2, e3 respectively). These events occur in the order they are numbered. The DF relationship between events e1 and e2, relates only to entity A and the DF relationship between events e1 and e3 only relates to entity B.

We define a *compound entity* as a set of entities, whose behaviour shall be studied together. For instance, consider BPI 14. incidents and configuration items each have their own, but possibly overlapping, behaviour. We can study their behaviour separately, but we might want to study how they behave together. To do this, we would define compound entities, each consisting of an incident and a configuration item.

Consider the compound entity $\{A, B\}$. This compound entity is related to all three events in our scenario, however we cannot immediately deduce the temporal ordering of the events with respect to this compound entity, as we cannot see whether event e2 or e3 directly follows event e1 and there is no DF relationship between events e2 and e3.

Since compound entities are not a concept in the original or new data model, how can an instance of such a model be enriched such that the temporal relations between events are modeled with respect to this compound entity?

Note that we do not intend to capture the behaviour of compound entities in the data model itself, as the type of compound entities that are useful for an analysis strongly depends on the question that analysis tries to answer. Therefore it makes more sense to enrich the data when required. Instead, we merely discuss how an instance of such a data model could be enriched to model the behaviour of compound entities.

In the original data model, since events nodes are allowed to have any number of incoming and outgoing DF relationships, we can simply order the events related to the compound entity by their start time and add DF relationships between LEvent nodes that don't already have a DF relationship, as is shown in Figure 3.13. Note that the added DF relationship is drawn using grey.

For the new data model, we model the temporal relations of compound entities as follows. We attach a new LEvent node to each event's Event node. We then connect all of those LEvent nodes to all Entity nodes that make up the compound entity using E_EN relationship. These new LEvent nodes are then connected with each other, using DF relationships, such that they form a single path of LEvent nodes, ordered by their start time as usual. This is shown in Figure 3.15. For each event *e1*, *e2* and *e3*, we create a new LEvent node *e1 (AB)*, *e2 (AB)* and *e3 (AB)* respectively. These LEvent nodes have a relationship to both Entity nodes A and B and have a relationship to their respective Event nodes. Finally, there is a path of DF relationships from *e1 (AB)* to *e2 (AB)* to *e3 (AB)*, to indicate their temporal relations with each other.

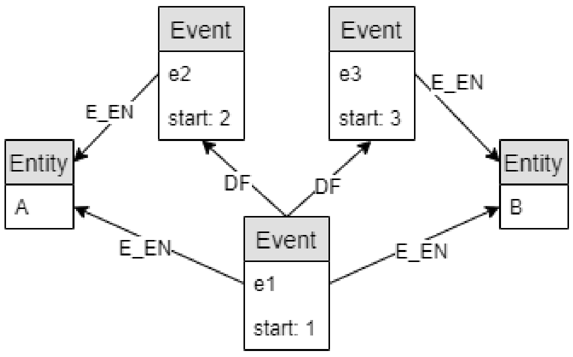

Figure 3.12: Compound entity scenario in the original data model

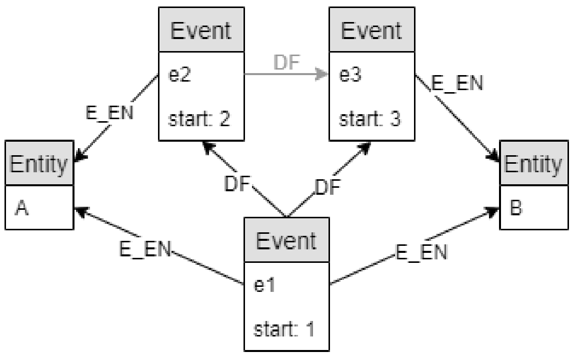

Figure 3.13: Representation of *DF* relationships for compound entity scenario in the original data model

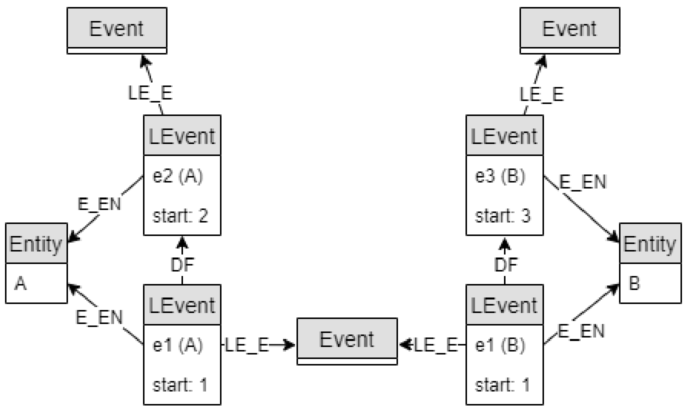

Figure 3.14: Representation of the data instance of Figure 3.12 in new data model

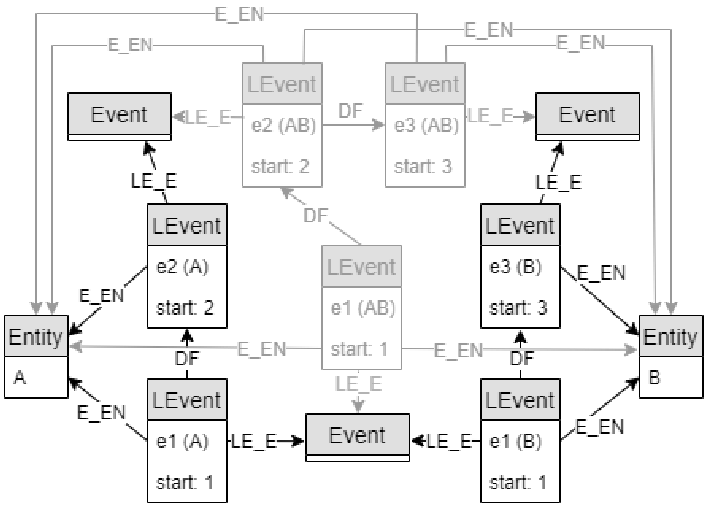

Figure 3.15: Representation of *DF* relationships for compound entity scenario in the new data model

Chapter 4

Data Transformation

In this Chapter we will discuss the transformation steps needed to go from relational event data to graph event data, as well as discuss the domain knowledge required to perform the transformation.

The first step is transforming the relational input data to a labeled property graph. For this we use R2PG-DM. For performance reasons, we had to make a conceptual change to the way R2PG-DM performs this transformation. In Section 4.1 we discuss this change.

After performing R2PG-DM's transformation, we have the relational input data, but represented in a generic graph data model. This generic graph data model does not contain the concepts of entities or events. Therefore we need to introduce these concepts to the graph data. We discuss this process in Section 4.2. This process starts with defining the desired conceptual model, using an EER model, which we introduce in Section 4.2.1. To create such an EER model, domain knowledge of the input data set is required. After an EER model has been created, we introduce the concepts we require for an event graph into the generic graph that was outputted by R2PG-DM. We do this in a number of transformation steps, described in Sections 4.2.2-4.2.5. After performing these transformation steps, we have an event graph of the input data, conforming to the data model we proposed in Section 3.2.7.

4.1 Relational data to labeled property graph

The first step of the transformation is to transform the relational data to a generic labeled property graph. In Section 2.4 we discussed how this transformation works. However, we have identified a performance issue with this transformation. Due to this performance issue, R2PGDM takes unnecessarily long to transform big datasets like BPI 14 and BPI 17.

This is why we had to make a conceptual change, as well as apply some implementation optimizations. We will only discuss the conceptual change in this section. Section 5.1 will discuss the implementation optimizations.

R2PG-DM transforms a relational database in two steps. The first step is the creation of nodes and their attributes from the input data. These nodes and attributes are stored in a new relational database in two separate tables, let's call these tables r_{nodes} and $r_{attributes}$ respectively. Each database row is transformed into one node and zero or more attributes. Each node thus corresponds with one database row of r_{nodes}. Each node record n contains the name of the corresponding row's table (accessible as $n.label$) and is assigned a number to uniquely identify it. Each attribute with a non-null value is assigned the same identifier as its corresponding node and is stored along with its name and value in the attributes table.

The second step is the creation of relationships. This step utilizes foreign keys between tables to determine which nodes should have a relationship to which. Let R be the set of tables of the input database. Let A_r be the set of attributes of $r \in R$.

Let AFK_{ij}, be the set of foreign key attribute pairs for two tables $i, j \in R$, such that $(a_i, a_j) \in AFK_{ij}$ if and only if $a_i \in A_i, a_j \in A_j$ and a_i references a_j via a foreign key. Let

CFK be the set of foreign keys of the input database, such that $(i, j, AFK_{ij}) \in CFK$ if and only if i has one or more foreign keys referencing j. For each $(i, j, AFK_{ij}) \in CFK$, relationships are created. Let N_i, N_j be the set of database records in r_{nodes}, that represent records of tables i, j respectively. Then relationship records between the to be created nodes of records of N_i and N_j are added to the output database in table r_{rels} as follows.

Each $(i, j, AFK_{ij}) \in CFK$ is processed sequentially:

1. For each $(a_i, a_j) \in AFK_{ij}$ we find all distinct tuples $(i, j, a_i.\text{name}, a_j.\text{name}, a_i.\text{value}, a_j.\text{value})$, where $a_i.\text{value} = a_j.\text{value}$.
2. For each of these tuples, we find all node records $S \subseteq N_i$ that have an attribute record named a_i.name with value a_i.value and all records $T \subseteq N_j$ that have an attribute record named a_j.name with value a_j.value.
3. Finally a relationship record is created for each pair of $S \times T$ and is stored in r_{rels}.

This way of creating of relationships is very inefficient, as it essentially performs a join (line 3) between the input database (line 1) and the output database (line 2). The input database is used to determine which rows reference each other via foreign keys; the output database is used to find the nodes that correspond to those rows.

To address this issue, we store the nodes, attributes and relationships tables in the input database. This way we can utilize the more efficient database engine to calculate the $S \times T$ set, using a single query per foreign key, as this way it has access to both the input and output data. In Section 5.1 we will discuss how exactly this query works.

After R2PG-DM's transformation we have tables r_{nodes}, $r_{attributes}$ and r_{rels}, which together describe a labeled property graph G. G can be created as follows. Each record of r_{nodes} represents one row of some table of the input database R; and each record of r_{nodes} is used to create a single node in G. Each record of the $r_{attributes}$ table represents one database cell of the input database R; and each record of $r_{attributes}$ also contains a node id referencing a record in r_{nodes}, in addition to the attribute itself. Every record of the $r_{attributes}$ table is added as an attribute to the node in G it references. Finally, every record of r_{rels} represents a foreign key relation between two database rows of R. For each record of r_{rels}, we add a relationship is between the two nodes in G that represent those two database rows, in the same direction as the foreign key relation.

G can be seen as a generic representation of the relational input data. However, this data does not conform to our data model, as it does not contain the concepts of events and entities.

4.2 Transforming labeled property graphs to the event data graph representation

Recall that after the transformation described in Section 4.1, we have a generic labeled property graph G. G has no concept of entities, events and the temporal relations between those events. In other words, we need to transform this generic labeled property graph G into a specific labeled property graph G', which conforms to the event graph data model we proposed in Section 3.2.7. By doing this, we explicitly introduce entities, events and temporal relations between events to G.

Before such a transformation can take place, we need to use some domain knowledge to define a schema that specifies how entities, events, and relations between entities and events can be found in the input data. This allows the transformation to access the input data in a structured way. We call such a schema an *EER diagram.* Section 4.2.1 discusses these EER diagrams. The input graph can then be transformed step-wise into the event graph representation. Each of Sections 4.2.2-4.2.5 discuss one of these transformation steps. The EER schema will introduce the concepts of *entity box*, *event schema* and *event* definition. Using these 3 concepts, we can define the transformation from G to G' by the following 4 types of transformation functions, as we will explain next.

- Creation of Entity nodes:
  $G.N \times$ *entity box* $\rightarrow$ Entity nodes
- Creation of events:
  $G.N \times$ *event Schema* $\times$ *event definition* $\rightarrow$ Event nodes $\times\ 2^{\text{LEvent nodes}} \times 2^{\text{LE\_E}} \times 2^{\text{E\_EN}}$
- Creation of DF relationships:
  Entity node $\times\ 2^{\text{LEvent nodes}} \rightarrow 2^{\text{DF}}$
- Creation of the Log node:
  LEvent nodes $\rightarrow$ Log node $\times$ L_LE relationships

Our transformation will not feature the handover of work (HOW) relationships, as we wanted to focus on just the temporal and structural aspect of the data model and these HOW relationships fall outside these aspects. However, these relationships could be added as an extra step at the end of the transformation, as all the information needed to calculate these relationships exists in the graph.

4.2.1 Defining desired conceptual model

There is no one way event data must be transformed into the event data graph representation. Some design choices need to be made per data set, by someone with domain knowledge. Not all tables in a database contain entities, or event data; and event data from one table may relate to entities in any number of other tables. This information needs to be provided to the transformation, in addition to the input data.

In order to visualize this information, we designed the Event Entity Relation diagram (EER diagram), which is used as the model for the transformation to event data graphs. Figure 4.1 shows a meta model for such diagrams. Figure 4.4 shows an instance of such a model, which describes a transformation of the BPI 14 data set. Figure 4.5 shows another instance of such a model, which describes a transformation of the BPI 17 data set.

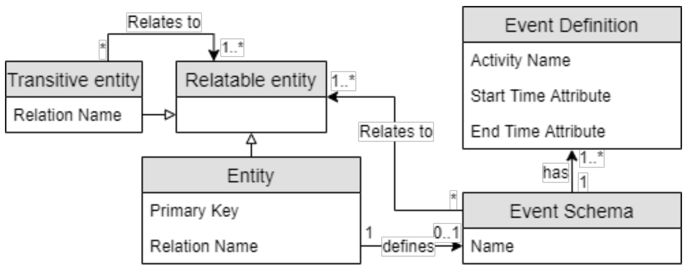

Figure 4.1: Meta model of the Event Entity Relation Diagram

First off, in Figure 4.4, there are the boxes with a grey header. These correspond to the *entity* box of the meta model in Figure 4.1. Also, recall that we use the term *entity* rather loosely, e.g., a table with events, such as BPI 17's 'application_events' table is considered to contain entities, even though they are technically events. Each *entity box* corresponds to one database table, but not all database tables have to be represented by an *entity box*. Tables from which records should not be included as Entity nodes in the graph G' should not be represented by an *entity box*. Let ENT be the set of *entity* boxes. Each *entity* box $ent \in ENT$ shows the name of the database table $ent.name$, and the name of that table's primary key $ent.key$. This information is used to create and uniquely identify Entity nodes in the graph database. We can use this to uniquely identify Entity nodes, because by our input data assumptions, each entity type is represented by one table and each row describes exactly one entity. In Figure 4.4, we define nine *entity* boxes

Next, in Figure 4.4, there are boxes with a white header. These boxes are called *event schema* boxes. Let ES be the set of *event schema* boxes. Each $es \in ES$ is owned by a single *entity* box $ent \in ENT$, which is shown using a solid arrow. Every *entity* box can own at most one *event schema* box. In other words $ent.es = es$ and $es.ent = ent$. In Figure 4.4, we define five *event schema* boxes

Each es has one or more *event definitions* $(act, start, end, es) \in es.ED$, which are situated in each *event schema* box, as can be seen in Figure 4.4. Each *event definition* $ed \in es.ED$ consists of four attributes. We call $es.ent$ the defining entity of es, which means that the column names of the table $es.ent.name$ can be used to assign values to these four attributes. The defining entity (which can be any node in ENT) of an event based on es, is the entity, whose attributes were used to assign values to the event's attributes. The activity name act can be described by an attribute of the defining entity or a string, denoted using quotation marks. The start time $start$ and end time end attributes are both defined using the defining entity's attributes.

For example, the event schema es of the 'Incident_Activity_Events' event schema box has $es.ent =$ Incident_Activity. The only event definition $ed = (act, start, end, es) \in es.ED$ has the following attribute values.

- $ed.act =$ IncidentActivity_Type
- $ed.start =$ DateStamp
- $ed.end =$ DateStamp
- $ed.es = es$

'IncidentActivity_Type' and 'DateStamp' are thus columns of the table named $ed.es.ent.name$. Note that 'Incident_Activity' is not surrounded by quotes in this *event definition*, in contrast to the *event definitions* of the 'Change_Events' *event schema* box, which define the activity name using a string. Since an entity can contain multiple attributes that contain time, multiple *event definitions* can be included in each *event schema* box. In the 'Incident_Activity_Events' *event schema* box of Figure 4.4, we define four *event definitions*. Also note that for BPI 17, only the "artificial" entities of which the entity type ends in '_events' have event schemas.

Next, In Figure 4.4, there is one diamond. We call this diamond a *transitive entity* box. *Transitive entity* boxes allow us to relate *event schema* boxes, to *entity* boxes that are more than one foreign key relation away from that *event schema* box's defining *entity*. Their exact function will become clear when we discuss the function of dashed lines. Each *transitive entity* box contains just the name of a table of the input database. A *relatable entity* box is then either just an *entity* box or a *transitive entity* box. Let TE be the set of *transitive entity* boxes.

Each *event schema* box $es \in ES$ is connected to zero or more *relatable entity* boxes using a dashed line. Let $RE = ENT \cup TE$ be the set of *relatable entity* boxes. These dashed lines indicate to which types of entities events created using any *event definition* $ed \in es.ED$ should relate.

Each event definition ed is owned by the entity box $ent = ed.es.ent$ (connected by a solid arrow). That means, for each entity node n of the type $ent.name$, we will create an event node e of type $ed.name$, but as each event can be related to more entities (See Sections 3.2.5-3.2.7), the event node e may have to relate to more relatable entity nodes (which are all entities of the types specified by the *relatable entity* boxes). Section 4.2.3 discusses this more precisely.

Consider any $es \in ES$, then let $es.related.ENT$ be the set of entity boxes connected via a dashed line and $es.related.TE$ be the set of transitive entity boxes connected via a dashed line. Each *transitive entity* box $te \in es.related.TE$ can in turn refer to other entities $te.ENT$ and other *transitive entities* $te.TE$ via dashed lines. It is not allowed to create a loop via these structures, it must be that $|te.ENT| > 0$ and that at some point along each path $te.TE = \emptyset$, so that this tree of $.ENT$ and $.TE$ steps, only has entity boxes at the leafs.

As an example, take es = 'Change_Events'. Using this event schema, we have $es.ENT = \emptyset, es.TE = \{$'Change_Activity'$\}$. If we now take te = 'Change_Activity', we have $te.ENT = \{$'Configuration_Item', 'Service_Component'$\}, te.TE = \emptyset$. At the leave of the tree of es, we have *Entity* boxes 'Configuration_Item' and 'Service_Component'. This tree of es describes how the entities at its leaves should be found in G, which is the graph created by R2PG-DM.

Now, take a look at Figures 4.2 and 4.3. where we see an example EER schema and an example generic property graph respectively. The EER schema in Figure 4.2 states that an event created from a D entity should relate to A entities which relate to that D entity via a C entity. Only the circled A entity in Figure 4.3 meets this requirement. The other A entities do not meet this requirement.

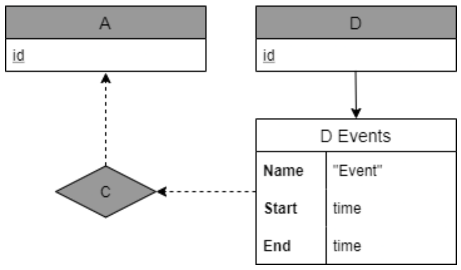

Figure 4.2: Example EER schema

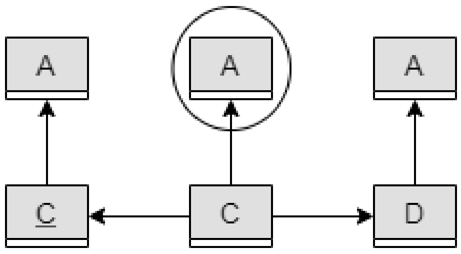

Figure 4.3: Demonstration of dashed line usage

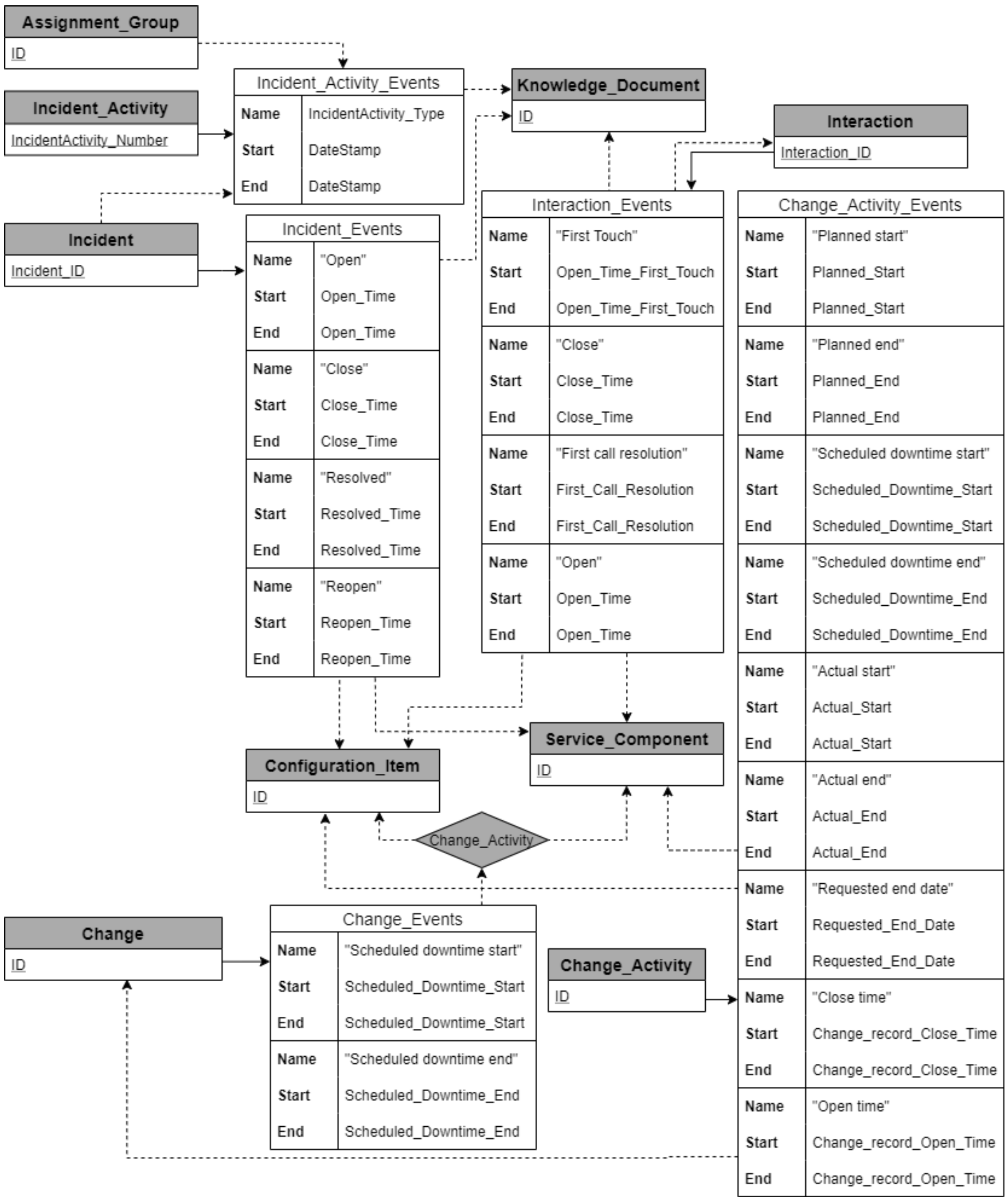
Assignment_Group
ID
Incident_Activity
IncidentActivity_Number
Incident
Incident_ID
Incident_Activity_Events
Name IncidentActivity_Type
Start DateStamp
End DateStamp
Incident_Events
Name "Open" Start Open_Time End Open_Time
Name "Close" Start Close_Time End Close_Time
Name "Resolved" Start Resolved_Time End Resolved_Time
Name "Reopen" Start Reopen_Time End Reopen_Time
Knowledge_Document
ID
Interaction
Interaction_ID
Interaction_Events
Name "First Touch" Start Open_Time_First_Touch End Open_Time_First_Touch
Name "Close" Start Close_Time End Close_Time
Name "First call resolution" Start First_Call_Resolution End First_Call_Resolution
Name "Open" Start Open_Time End Open_Time
Change_Activity_Events
Name "Planned start" Start Planned_Start End Planned_Start
Name "Planned end" Start Planned_End End Planned_End
Name "Scheduled downtime start" Start Scheduled_Downtime_Start End Scheduled_Downtime_Start
Name "Scheduled downtime end" Start Scheduled_Downtime_End End Scheduled_Downtime_End
Name "Actual start" Start Actual_Start End Actual_Start
Name "Actual end" Start Actual_End End Actual_End
Name "Requested end date" Start Requested_End_Date End Requested_End_Date
Name "Close time" Start Change_record_Close_Time End Change_record_Close_Time
Name "Open time" Start Change_record_Open_Time End Change_record_Open_Time
Configuration_Item
ID
Service_Component
ID
Change_Activity
Change
ID
Change_Events
Name "Scheduled downtime start" Start Scheduled_Downtime_Start End Scheduled_Downtime_Start
Name "Scheduled downtime end" Start Scheduled_Downtime_End End Scheduled_Downtime_End
Change_Activity
ID

Figure 4.4: event entity relation diagram of BPI 14 data set

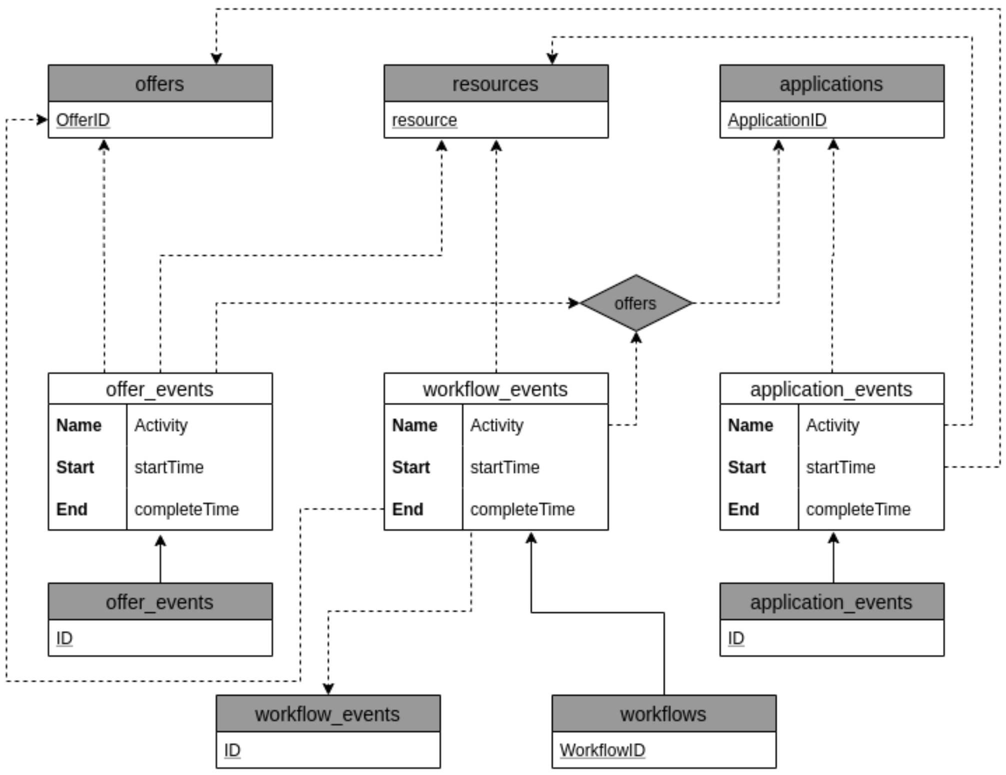

Figure 4.5: event entity relation diagram of BPI 17 data set

4.2.2 Creating entities

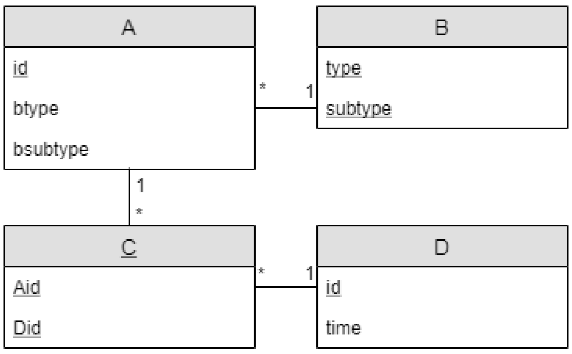

Figure 4.6: Example ER model

A		
id	**btype**	**bsubtype**
1	type 1	subtype 1
2	type 2	subtype 1

B	
type	**subtype**
type 1	subtype 1
type 2	subtype 1
type 2	subtype 2

C	
Aid	**Did**
1	1
1	2
2	1

D	
id	**time**
1	1
2	2
3	1

Table 4.1: Example data following the ER model shown in Figure 4.6

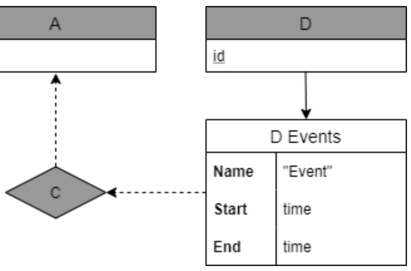

Figure 4.7: Possible event entity relation diagram of example ER model and data of Figure 4.6 and Table 4.1

Now that we have an EER diagram, we need a method to transform the nodes and relationships from R2PG-DM's output to the event graph data model. To illustrate this transformation we will be transforming the data from Table 2.6 with its ER model shown in Figure 2.5. In Chapter 2 we also used this example data set. We have constructed an EER model for this dataset, which is shown in Figure 4.7. After each transformation step, we will show the resulting graph with this example data as input data.

The first step of this transformation is creating *Entity* nodes. This transformation step can be described by the function $G.N \times \mathit{entity\ box} \rightarrow \mathsf{Entity}$ nodes. Let I_n be the set of nodes present in the graph as a result of R2PG-DM's transformation. Let $I_r \subseteq {I_n}^2$ be the set of relationships between those nodes. Let $N_{entity} = \emptyset$ be the set of Entity nodes. First, for each input node $n \in I_n$, whose label $n.label$ is in $\{ent.name | ent \in ENT\}$, we add an Entity node e to N_{entity}. Let ent_n be the entity box for which $e.name = n.label$. Then e gets the following attributes:

- $e[EntityType] = n.label$
- $e[IDraw] = n[ent_n.key]$

- $e[IDlog] = n.label + n[ent_n.key]$
- $e[p] = n[p]$ for any other entity attribute p

Note that determining what are entity attributes and what are event attributes are left to the implementation. The resulting graph after performing this transformation step on our example data is shown in Figure 4.8.

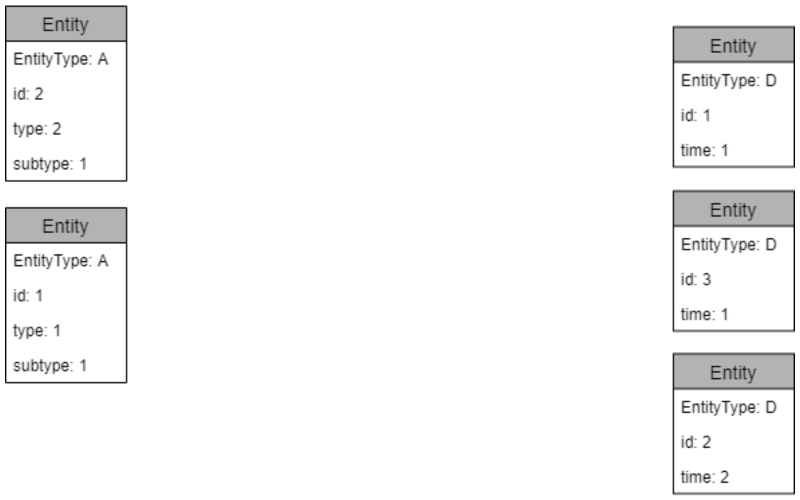

Figure 4.8: Event graph after creating Entity nodes

4.2.3 Creating events and relating them to entities

Now that we have created Entity nodes, we go on to the next transformation step, which is to create LEvent and Event nodes, described by the function $G.N \times \textit{event Schema} \times \textit{event definition} \rightarrow$ Event nodes $\times\, 2^{\text{LEvent nodes}} \times 2^{\text{LE\_E}} \times 2^{\text{E\_EN}}$. Each *event schema* box $es \in ES$ has one or more *event definitions* $ed \in es.ED$. For each es, take the set $I_{es} = \{n | n \in I_n \wedge n.label = es.ent.name\}$. This is the subset of the input nodes I_n that correspond to the entity type $es.ent.name$ that defines the *event schema* box es.

Each event may relate to multiple entities (not just one entity of the entity type that 'owns' the event definition ed, but also entities of other types). For example, an event created using the event definition of BPI 14's 'Incident_Activity_Events' event schema may relate to an 'Incident' entity, an 'Assignment_Group' entity and a 'Knowlege_Document' entity in addition to an 'Incident_Activity' entity (Which is the entity type that owns that event definition). We thus need to find for each event e that will be created using ed, to which set of entities event e should relate. For this we use $es.related$, i.e., the set of all entities reachable from the *event definition* box via dashed lines. For each *event definition* $ed \in es.ED$ we do the following

1. For each node $n \in I_{es}$, we create one Event node e. Event node e can relate to multiple entities, so we need to find the set of entities $I_{re(n)} \subseteq I_n$ to which e should relate. To construct $I_{re(n)}$, we find all nodes $n' \in I_n$ that structurally relate to n according to the dashed line tree of $es.related$. Algorithm 1 describes how the dashed line tree of $es.related$ is used to find which nodes belong to $I_{re(n)}$. Algorithm 1 is called as traverseTETree($es.related, \{n\}$) for each n.

Algorithm 1: traverseTETree

```
input :
    C: related attribute of an event schema box or transitive entity box
    T: set of nodes of G (should contain only one node in the initial call to this algorithm)
output: A set of nodes that relate to nodes in T according to C
begin
    $R \leftarrow \emptyset$
    /* Add all nodes with a label of any entity box in C.ENT, that can be
       reached from any node in T via one relationship                    */
    $R \leftarrow R \cup \{t | t \in I_n \wedge \exists_s[s \in \mathbf{T}, (s,t) \in I_r \wedge \exists_{ent \in \mathbf{C}.ENT}[t.label = ent.name]]\}$
    for $te \in \mathbf{C}.TE$ do
        /* Calculate a set of nodes with a label of any entity box in te,
           that can be reached from any node in T via one relationship    */
        $T_{te} \leftarrow \{t | t \in I_n \wedge \exists_{s \in \mathbf{T}}[(s,t) \in I_r \wedge t.label = te.name]\}$
        /* te, together with the set $T_{te}$, forms the input to the next
           recursive call to this function. $T_{te}$ is the set of nodes that
           were reached by following the dashed lines of the EER schema from
           the original call's C to te. The next recursive call will then
           expand $T_{te}$ using te.related to find more nodes to add to R      */
        $R \leftarrow R \cup$ traverseTETree($te, T_{te}$)

    return R
```

2. For each node $n \in I_{es}$ we do the following:

   i For each $i_{re} \in I_{re(n)}$ we create an LEvent node le which all belong to the Event with the following attributes:

   - $le[Activity] = ed.name$ if $ed.name$ is a string
   - $le[Activity] = i_{re}[ed.name]$ if $ed.name$ is an attribute name

- $le[Start] = i_{re}[ed.start]$
- $le[End] = i_{re}[ed.end]$
- $le[p] = i_{re}[p]$ for any event attribute p of i_{re}

ii We create an LE_E relationship between each le and e, to indicate that all these LEvent nodes we just created represent the same event. We also create an E_EN relationship between each LEvent le and the Entity node that was created to represent i_{re}. This Entity node's *IDlog* attribute is equal to $n.label + n[es.ent.key]$ (which is how we the defined the *IDlog* attribute of the Entity node created from the input node n). To summarize, for the set I_{re}, we have created one Event node and one LEvent node per element of that set, each connected to that Event node and one Entity node. The resulting graph after performing this transformation step on our example data is shown in Figure 4.9.

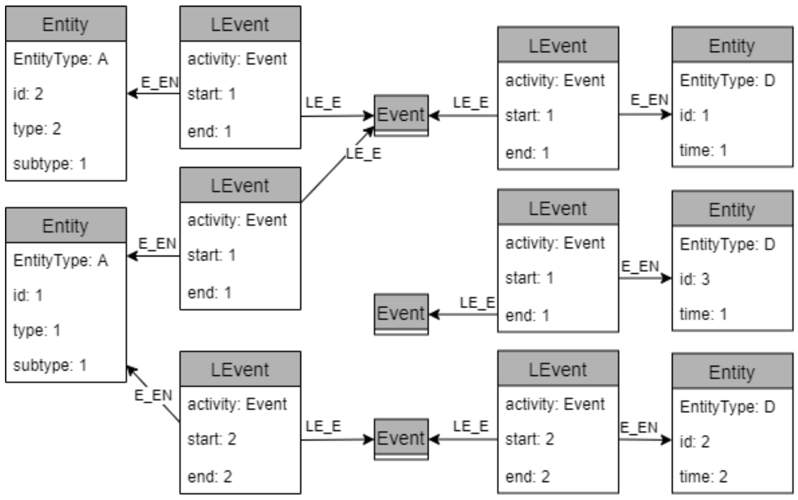

Figure 4.9: Event graph after creating Event and LEvent nodes and relating them to Entity nodes

4.2.4 Calculating directly follows relations

Now that we have created all Entity, Event and LEvent nodes, we can begin with the next transformation step, which is calculating DF relationships between LEvent nodes, as described by the function Entity node $\times\, 2^{\mathsf{LEvent\ nodes}} \rightarrow 2^{\mathsf{DF}}$. From this point onwards we don't requrie the EER schema anymore. These DF relationships, are calculated per Entity node $e \in N_{entity}$. For each e, find all event nodes EV connected to e via an E_EN relationship. Now we sort EV by the start time attribute of LEvent nodes resulting in a list $[ev_1, ev_2, \ldots, ev_k]$, such that the LEvent node with the earliest start time is the first element. In case two events of EV have the exact same start time, their ordering has to remain consistent with the ordering of LEvent nodes connected to the same Event nodes as those two events, when calculating the 'directly follows' relationships for all other entities.

Now, between each of the nodes $ev_i, ev_{i+1} \in EV, 0 \leq i < |EV| - 1$ we create a DF relationship. As a result, the event nodes of EV form a path, where the first node in the path has the earliest start time, and the final node has the latest start time. The resulting graph after performing this transformation step on our example data is shown in Figure 4.10.

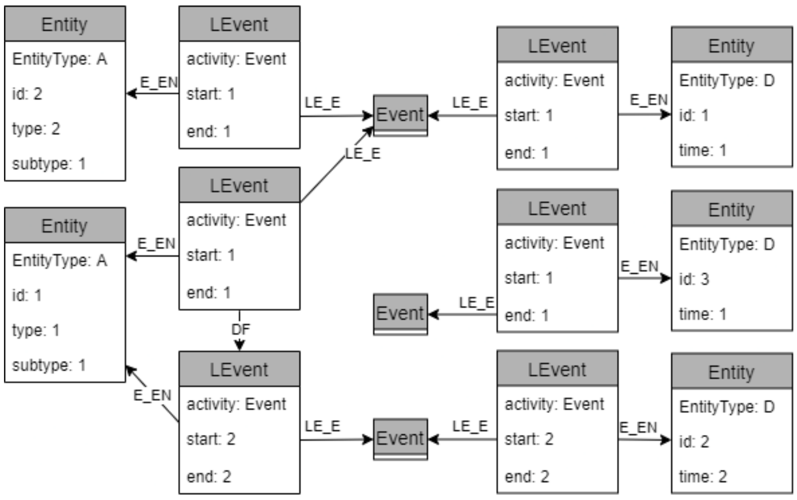

Figure 4.10: Event graph after creating DF relations

4.2.5 Creating the Log node and relating it to events

The last step in the transformation is the creation of the Log node, which is described by the function LEvent nodes → Log node × L_LE relationships. We create a single Log node and connect it to all event nodes via a L_E relationship, after which the transformation has been completed. The resulting graph after performing this transformation step on our example data is shown in Figure 4.11.

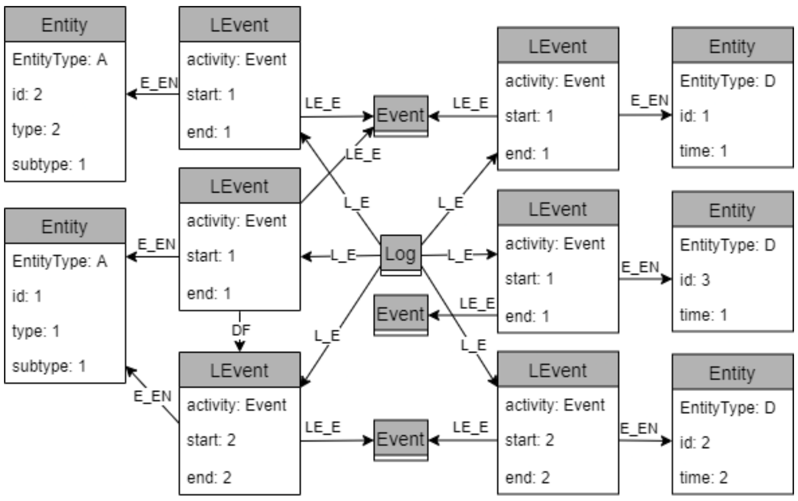

Figure 4.11: Event graph after creating the Log node and relating it to LEvent nodes

Chapter 5

Implementation

In this Chapter we will discuss the implementation of the transformation steps needed to go from relational event data to graph event data that we discussed in Chapter 4. In Section 5.1, we start by discussing the implementation of the changes we made to R2PG-DM.

In Section 5.2 we discuss the implementation of the transformation of generic property graphs to event graphs, which we discussed in Section 4.2. We start by discussing the configuration of the transformation in Section 5.2.1. This configuration is a manifestation of the EER model we described in Section 4.2.1. Following the configuration, in Sections 5.2.2-5.2.6 we describe the implementation of a series of transformation steps that introduce the event graph concepts to the generic property graph. We discussed these steps conceptually in Sections 4.2.2-4.2.5. After performing these transformation steps, we have an event graph representation of relational event data in a Neo4J database instance.

5.1 Relational data to labeled property graph

In addition to the conceptual change to R2PG-DM, which we discussed in Section 4.1, we also made some other changes to its implementation in order to reduce its processing time. We also found and fixed a bug that caused faulty relationships to be created.

Section 5.1.1 discusses the implementation of the conceptual change to R2PG-DM we discussed in Section 4.1. Sections 5.1.2 and 5.1.3 discuss two other changes we made to R2PG-DM's implementation. In Section 5.1.4 we discuss the implementation of the transformation from R2PG-DM's output to a data model that allows for faster bulk importation into Neo4J.

5.1.1 Implementation of conceptual change

First, lets come back to the conceptual change. As we mentioned before, instead of writing the output of R2PG-DM to a new database, we write it to the input database, as described in Section 4.1. This allows us to replace the joins between input and output data that R2PG-DM performs in memory to create relationships, by joins performed by the database engine. The downside of this is that the input database is not allowed to have tables with the name 'node', 'property' or 'edge', as the program would overwrite those tables. The creation of relationships is still done separately for each foreign key, but instead, we create all relationships for that foreign key using a single query. Each executed query thus creates all relationships between nodes from two tables in the output graph G.

Recall that before relationships are created, nodes and node attributes have already been created and stored in two separate output tables r_{nodes} and $r_{attributes}$ (see Section 4.1). Each query handles one foreign key, and thus handles a source and a target table of that foreign key.

Looking at Table 5.1, the R2PG-DM translation will create 2 nodes for the 2 records in table A (n0,n5 in Figure 5.2) and 3 nodes for the 3 records in table B (n1,n4,n8).

We first need to combine the tables r_{nodes} and $r_{attributes}$, such that we pair each node record n with attribute records that reference node record n's node id. We do this for each node record $n_i \in r_{nodes}$ with $n_i.label$ equal to the name of the source table and for each node $n_o \in r_{nodes}$ with $n_o.label$ equal to the name of the target table separately. We then pivot both these tables such that their column names are equal to the foreign key attribute names. We can then join these two tables to find which source nodes should have a relationship to which target nodes.

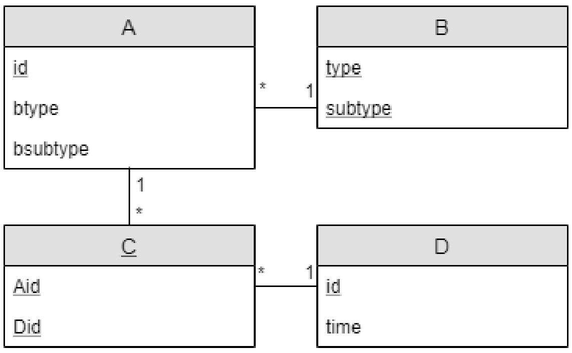

Figure 5.1: Example ER model

A		
id	**btype**	**bsubtype**
1	type 1	subtype 1
2	type 2	subtype 1

B	
type	**subtype**
type 1	subtype 1
type 2	subtype 1
type 2	subtype 2

C	
Aid	**Did**
1	1
1	2
2	1

D	
id	**time**
1	1
2	2
3	1

Table 5.1: Example data following the ER model shown in Figure 5.1

In our case, we used SQLite as our relational database engine. To show how the query works, consider the tables A and B and their foreign key relation, shown in Figure 5.1 and Table 5.1. We thus need to transform the tables A and B of Table 5.1; and tables r_{nodes} and $r_{attributes}$ in such a way that we can:

a) Link each node that was created for a record to the record itself (e.g. link the record with $id = 1$ in table A to node $n0$, which is the node in Figure 5.2 with nodeID = 0)

b) Identify which concrete records (and their related nodes) of A are related to which concrete records (and their related nodes) of B (e.g., that the node $n0$ created from the record (1,type1,subtype1) in A is related to the node $n5$ created from the record (type1,subtype1) in B

Listing A.1 shows the query that creates relationships between the nodes created from tables A and B. As can be seen in this listing, the query is built in parts. Therefore we will explain what each part does starting at the 'joinableColumns' part.

The 'joinableColumns' subquery contains all combinations of values in the foreign key columns between the source and target tables that occur in both the source and target table. Here, table A is our source table and table B is our target table. table A references table B via two foreign key columns: *btype* and *bsubtype*, which reference table B's *type* and *subtype* columns respectively. The resulting table is shown in Table 5.2. This table contains all the values of table B that occur in the foreign key relation to table A.

Next is the 'sourceNodes' subquery. This subquery finds all (id, key, value) triples of nodes of the source table that have the foreign key columns as attributes. I.e., it finds each record in table

btype	bsubtype
type 1	subtype 1
type 1	subtype 2

Table 5.2: joinableColumns

A, the corresponding nodes n_0 and n_5 from table r_{nodes} and the corresponding attributes and values that define the foreign key relation from $r_{attributes}$. Figure 5.2 shows the node identifiers we have assigned to nodes created from tables A and B as the 'nodeID' attribute on each node. The 'targetNodes' subquery does the same, but for the target table. The resulting tables are shown in Table 5.3.

The next step is to pivot the 'sourceNodes' table, which is what the 'pivotedSourceNodes' subquery does. It pivots the 'sourceNodes' table on the 'pkey' column, such that the values in this column, which are exclusively the foreign key column names, become the column names, with their value equal to the corresponding 'pvalue'. The resulting tables are shown in Table 5.4. The same is done with 'targetNodes'.

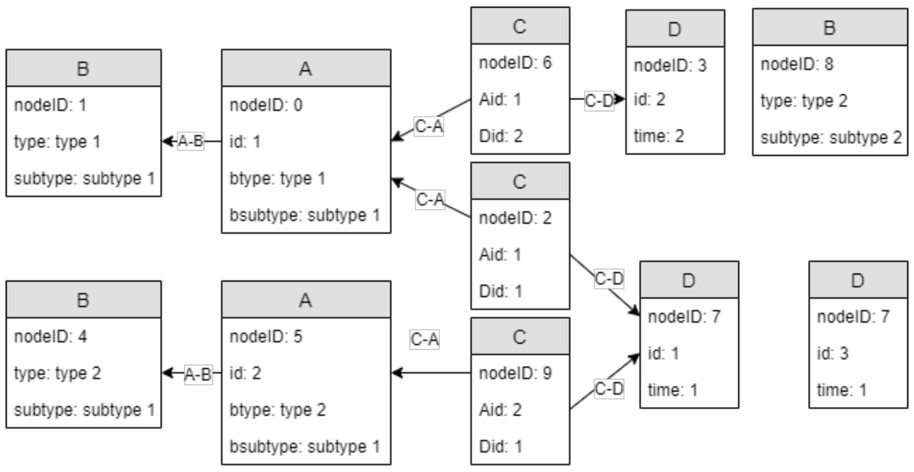

Figure 5.2: Property graph as a result of R2PG-DM transformation of data shown in Table 2.3

id	pkey	pvalue
0	btype	type 1
0	bsubtype	subtype 1
5	btype	type 1
5	bsubtype	subtype 2

id	pkey	pvalue
1	type	type 1
1	subtype	subtype 1
4	type	type 2
4	subtype	subtype 2
8	type	type 1
8	subtype	subtype 2

Table 5.3: sourceNodes and targetNodes

id	btype	bsubtype
0	type 1	subtype 1
5	type 1	subtype 2

id	btype	bsubtype
1	type 1	subtype 1
4	type 2	subtype 2
8	type 1	subtype 2

Table 5.4: pivotedTourceNodes and pivotedTargetNodes

Now, an inner join is performed on the ‘pivotedSourceNodes‘ and ‘joinableColumns’. The resulting table is shown in Table 5.5.

id	btype	bsubtype
0	type 1	subtype 1
5	type 1	subtype 2

Table 5.5: joinedSourceNodes

Finally, ‘joinedSourceNodes’ is left joined with ‘pivotedTargetNodes’, which gives us all source-target pairs of node ids from which we can create relationships. We perform a left join to eliminate any node records from table B that should not be related to a node record from A. The result is shown in Table 5.6. The result of this table can then be stored in the ‘edge’ table with type equal to ‘A-B’.

sourceID	targetID
0	1
5	4

Table 5.6: relationships

5.1.2 Query batching and multithreading

R2PG-DM has a long processing time for a couple of other reasons. First, it only used a single thread, while most modern computers have multiple available. Second, each node, attribute and relationship record is written using a separate database transaction, which adds a lot of extra overhead.

We solved the first problem by handling each table of the input database in a separate thread. During the creation of nodes and attributes of each table, no data produced using other tables is required, so they can be processed in parallel. During the creation of relationship records of each table, node and attribute records from other tables are required, but those records are already present, as long as the node and attribute record creation finishes before starting the creation of relationship records. Thus we can also run the creation of relationship records of each table in parallel.

The second problem was solved by batching the creation of node and attribute records; and batching the creation of relationship records. Instead of sending a transaction for each to the database, we group a large amount of insertion queries into one query, such that the data doesn't exceed the transaction size limit. For this we used the transaction size limit of SQLite. However, the transaction size limit may have to be adjusted for other relational database engines. This way we greatly reduce the overhead that was caused by not grouping insertion queries.

5.1.3 Bug in relationship creation

As mentioned before, we found a bug in the implementation of R2PG-DM. This bug caused faulty relationships to be created when a table has foreign keys that refer to multiple tables. This was because the relationships were created per two foreign keys, instead of per foreign key. The program looped over the foreign keys, skipping every other foreign key. This was likely an error caused by a wrong assumption of the program's creator, which was not caught since it worked for the databases that were used to test R2PG-DM originally. We fixed this issue by looping over the foreign keys instead.

5.1.4 Preparing the output data to be used by Neo4J's admin import

As we discussed in Section 2.4, the output of R2PG-DM is meant to be generic. Neo4J can import this data directly using R2PG-DM's output format, however this way of importing data is very slow. Luckily, Neo4J has another method to import data: The 'admin import'. This admin import however requires a different format than what R2PG-DM provides and can only be applied to fresh database instances (Database instances in which no data was ever stored).

Figure 2.4 shows the three relations that form the output of R2PG-DM. These relations need to be transformed and stored in *.csv* files.

In this new format, instead of having node and attribute records in separate *.csv* files, we need to merge the files, such that there is a *.csv* file for the node and attribute records of each table in the input database. Each *.csv* file has multiple columns, one per column of the table it represents, as well as a column with the name ':ID' and a column with the name ':LABEL' which contains the node id and node type for each node record respectively.

Since R2PG-DM doesn't use any relationship attributes, we only need one *.csv* file containing relationships. this *.csv* file, like the relationships file outputted by R2PG-DM, has four columns. The only change we need to make is to rename the 'srcId' column to ':START_ID', the 'tgtId' column to ':END_ID' and rename the 'label' column to ':Type', which are the column names that Neo4J expects to find.

5.2 Transforming labeled property graphs to the event data representation

Before we discuss the implementation of our transformation, there is another issue that we'd like to address, which is that storing event attributes is very expensive. In our implementation, LEvent nodes are created from the same data as entities. Recall that in Section 3.1, we required each database row to only contain one entity. Event attributes are expensive to store, because the data of one entity can be used to create multiple events and thus event nodes. Each of these event nodes would then store the same event attributes from that entity, resulting in data duplication. We'd like to eliminate this.

To do this, we no longer separate the event attributes from the entity attributes and instead store all attributes on the Entity node. LEvent nodes keep only the minimally required attributes ('start', 'end' and 'activity'). The graph should then be interpreted as follows. Consider any Entity node e in the graph. All attributes of e are regarded as that entity's entity attributes. Now take any LEvent node le, `(e)<-[:E_EN]-(le)`, then all attributes on Entity nodes $e' \in E_{ev}$, `(le)-->(ev:Event)<--(:LEvent)-[:E_EN]->(`E_{ev}`)`, which are the Entity nodes that also have event ev, are considered the event attributes of Event ev from e's perspective (Note that $e \notin E_{ev}$).

This approach comes with the downside that, when looking at an entity, you can't directly tell what the value of an event attribute is supposed to represent. It could be the first value that was set when the entity was created, or the last value it was assigned, or any value in between. E.g. if at some point it was decided to no longer update that value in that entity's table, but instead in a separate table. This problem does not occur if the entity originates from a table that records for example changes, as then it can be assumed the database row was created and left unchanged afterwards. A simple solution to alleviate this problem would be to change the schema of the input database such that all event attributes are stored in a separate table such that that table only records changes to the entities in the original table.

Now, to come back to the implementation of this transformation. We implemented this transformation in Python3. It is meant to be used with a Neo4J v3.* database and might not work otherwise. In the upcoming sections we will discuss how to configure the transformation program and the various transformation steps.

5.2.1 Configuration

The configuration is a manifestation of the EER schema we discussed in Section 4.2.1. The configuration is created by the user and will be used during the transformation to generate Cypher queries. We store this configuration c in a *JSON* format. We start out with an empty root object: {}. At this root we have an object *c.connection* to specify how to connect to the neo4j database instance containing R2PG-DM's output data for the to be transformed data set. We also have a *c.log* object, which specifies this log's name. Then finally we have an *entity* array *c.entity*, which stores all the information present in the EER diagram.

Each *entity* object $c_e \in c.entity$ represents one *Entity* box $ent \in ENT$ of the EER schema. This *entity* object has three attributes:

- $c_e.label = ent.name$
- $c_e.id\_column = ent.key$
- $c_e.event$: object describing the event schema $ent.ES$

The *event* attribute should be omitted if no events should be created from entities with type $c_e.label$. Each event schema *event* has an attribute *create_from*, which is an array containing that event schema's event definitions $ent.ES.ED$. Each event definition $c_{ed} \in event.create\_from$ represents one event definition $ed \in ent.ES.ED$ and is an object with three attributes:

- $c_{ed}.start\_column = ed.start$
- $c_{ed}.end\_column = ed.end$
- $c_{ed}.activity$, based on $ed.act$

For the event definition's activity attribute we allow a bit more flexibility than we do in the EER schema. The activity attribute consists of a single string. This activity name of LEvent nodes created from this event definition is then equal to this string. However, within this string you can substitute parts with the values from attributes of Entity nodes with type $c_e.label$, by writing the name of the attribute for which you want to substitute values between brackets.

For instance, if we have an *Incident_Activity ia* with *ia.Incident_Type = Start* and we configure the *activity* attribute as *"Incident_Activity: {Incident_Type}"*, then the activity name would be *"Incident_Activity: Start"*

$c_e.event$ has another attribute, *related_entities* which represents $ent.ES.related$. This attribute contains a list of strings. Here we require the configuration to be somewhat more explicit than the dashed lines of the EER diagram, as we need to know the direction of the relationships to the related entities.

Lets take BPI 14's Change_Activity table as an example. Also recall Figure 2.8, which shows the ER diagram of our normalized BPI 14 data set. Change_Activity's event schema has a dashed line towards the Change entity box. Now, since we are considering the Change_Activity table and the Change table is referenced by the Change_Activity table, the string to represent this dashed line would be ">Change" (towards Change). If the Change_Activity table would have been referenced by the Change table instead, the string would be "<Change" (from Change). Furthermore, *related_entities* would contain the strings ">Service_Component" and ">Configuration_Item".

Now take BPI 14's Change table as an example. This table has a dashed line to a Change_Activity transitive entity box, which in turn has a dashed line to both the Service_Component and Configuration_Item entity boxes. A colon indicates that the entity type to the left of that colon is a transitive entity box. The *related_entities* attribute would then consists of the two strings "<Change_Activity:>Service_Component" and "<Change_Activity:>Configuration_Item".

Listing 5.1 shows the entity config of BPI 14's 'Change' and 'Service_Component entity types. This is a small part of the full BPI 14 transformation configuration, which can be viewed in Listing

B.1. BPI 17's transformation can also be viewed in Listing C.1. In the upcoming Sections, we explain how the JSON representation of the EER schema allows us to generate Cypher queries for each of the 4 steps we discussed in Section 4.2.

```
...
{
  "label": "Change",
  "id_column": "ID",
  "event": {
    "related_entities": ["<Change_Activity:Service_Component", "<
       Change_Activity:Configuration_Item"],
    "create_from": [
    {
      "start_column": "Scheduled_Downtime_Start",
      "activity": "Change: Scheduled_Downtime_Start"
    },
    {
      "start_column": "Scheduled_Downtime_End",
      "activity": "Change: Scheduled_Downtime_End"
    }
    ]
  }
},
{
  "label": "Service_Component",
  "id_column": "ID"
}
...
```

Listing 5.1: Entity Config op BPI 14's 'Change' and 'Service_Component' entity types

5.2.2 Creating entities

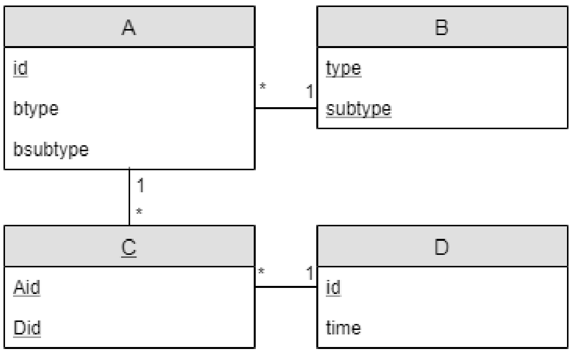

Figure 5.3: Example ER model

We again use our example database of Table 5.3 to show what the event graph looks like after each transformation step. Figure 5.4 shows the EER model we use for this transformation. During this transformation we will show a number of Cypher queries. These Cypher queries are from the BPI 14 transformation we performed. Listing 5.1 shows a part of the BPI 14 configuration file that was used to generate the queries we show.

A		
id	**btype**	**bsubtype**
1	type 1	subtype 1
2	type 2	subtype 1

B	
type	**subtype**
type 1	subtype 1
type 2	subtype 1
type 2	subtype 2

C	
Aid	**Did**
1	1
1	2
2	1

D	
id	**time**
1	1
2	2
3	1

Table 5.7: Example data following the ER model shown in Figure 5.3

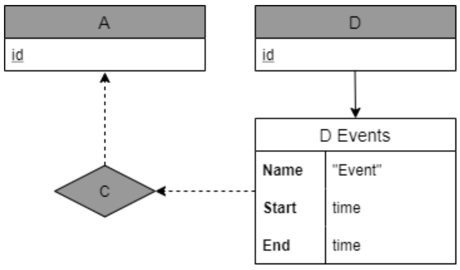

Figure 5.4: Possible event entity relation diagram of example ER model and data of Figure 5.3 and Table 5.7

The first nodes we create are the Entity nodes. We sequentially process the creation of entities per entity type that's listed in the configuration file. Recall that c is the configuration file root. Lets call the configuration object (entity type config) of the entity type that is currently being processed $c_e, c_e \in c.entity$. Recall that each $c_e \in c.entity$ represents one *entity* box $ent \in ENT$.

Consider Listing 5.2, which shows the query generated from Listing 5.1 that was used to create entities of the type 'Change'. This query is based on the EER diagram/configuration file of the BPI 14 transformation. The specific configuration used for Listing 5.2, can be seen in Listing 5.1.

We want to create an Entity node for each input node with label $c_e.label$. This is done as follows.

1. We create a matcher that simply matches all input nodes $n \in N$ with $n.label = c_e.label$. (line 1)

2. For each input node n we create an Entity node e. We use Neo4J's 'apoc' package to create these nodes, as it gives us a bit more flexibility over Neo4J's own node creation query. This Entity node e is assigned a number of attributes.

   - $e[EntityType] \leftarrow c_e.label$ (line 4)
   - $e[IDLog] \leftarrow c.log + n[c_e.id\_column]$ (line 5)
   - $e[IDraw] \leftarrow n[c_e.id\_column]$ (line 7)
   - $e[Log] \leftarrow c.log$ (line 8)
   - $e[uID] \leftarrow c_e.label + c.log + n[c_e.id\_column]$ (line 9)
   - $e[p] \leftarrow n[p]$ for any attribute p of input node n (line 11)

Figure 5.5 shows what the event graph of our example database would look like at this stage. The 'NodeID' attribute on each Entity node is the node id $ID(x)$ that uniquely identifies each node x in the graph. For clarity we don not show the 'IDlog', 'Log' and 'uID' attributes of Entity nodes.

```
1  MATCH (n:Change)
2  CALL apoc.create.node(
3     ['Entity'],
4     {
5        EntityType:'Change',
6        IDLog:'BPI14' + n.ID,
        IDraw: n.ID,
        Log:'BPI14',
        uID:'ChangeBPI14'+ n.ID
    }) yield node
    SET node+=n
```

Listing 5.2: Creation of Change entities

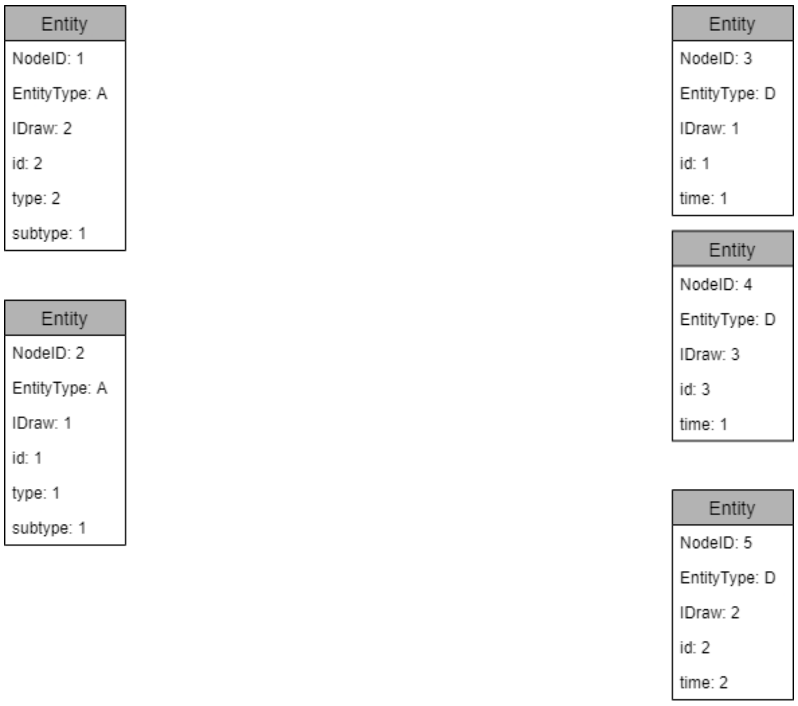

Figure 5.5: Event graph after creating Entity nodes

5.2.3 Creating events

After the creation of Entity nodes has finished, the next step is to create LEvent and Event nodes. Like with the creation of entities, we sequentially process the creation of events per entity type and thus entity configuration c_e. Not every entity configuration c_e specifies an event object $c_{ev} = c_e.event$. These entity types are simply skipped. In essence this means that we create events sequentially for each *event schema* box $es \in ES$, as every $c_e \in c.entity$ has at most one event schema associated with it. Let N be the set of nodes in G with label $c_e.label$. The event creation process can be split into two stages.

1. We find for each input node $n \in N$ all related nodes according to c_{ev} and create TempLEvent (Template Local Event) nodes for them. Each TempLEvent node represents an entity that relates to the event(s) of n. We create these TempLEvent nodes, since we need to know which entities are related to each event multiple times, so this way we don't have to calculate that each time, but instead just once at the beginning.

2. We process each event definition $c_{ed} \in c_{ev}.create\_from$, copy TempLEvent nodes to create LEvent nodes for c_{ed} and link those LEvent nodes to Event nodes. Every time this process is finished for one entity type, all TempLEvent nodes are deleted from the graph, as they have served their purpose and are no longer useful afterwards.

These two steps above are both performed before moving on to the next entity type. Now we will go into more details about these two steps.

Step 1: Creating Template LEvents

The first nodes for which we create TemplLEvent nodes are the input nodes in N, which are the nodes created from the table r_{nodes} by R2PG-DM. Listing 5.3 shows the query that does this for BPI 14's 'Change' nodes. This query was generated using lines 3-18 of the configuration in Listing 5.1. Recall that $ID(x)$ means the node id of node x, which is unique across all nodes in the graph. First, we match all input nodes N then, for each input node $n \in N$, create a TemplLEvent node te with $originID = ID(n)$ and $commonID = ID(n)$; and create a Source relationship from te to n. We also set the attributes shown on Lines 4-5 on the created node. The values of those attributes are based on $c_e.label$ and $c_e.id\_column$ again.

```
1  MATCH (common:Change)
2  CREATE (n:TemplLEvent {originID: ID(common), commonID: ID(common)})
3  -[s:Source]->(common)
4  ON CREATE SET n.EntityType="Change"
5  ON CREATE SET n.IDraw=common.ID
```

Listing 5.3: Creation of Change template events

Recall that each event may relate to multiple entities other than an entity of the type $c_e.label$ that owns the *event definition* that owns that event (Section 4.2). We create TemplLEvent nodes for those other input nodes of N (which are the input nodes with with label $c_e.label$). For each $n \in N$ we find the set of other input nodes that relate to events of n. This set is called $I_{rel(n)}$, as we discussed in Section 4.2. For every $n \in N$, we create a TemplLEvent node for every $m \in I_{rel(n)}$. Every TemplLEvent node thus represents a pair of input nodes (n, m). Let T be the set of all (n, m) pairs. Each related entity type string $c_{rel} \in c_{ev}.related\_entities$ specifies how we can find a subset $T_{rel} \subseteq T$, of all related entities of the type specified by c_{rel}. If we add all sets T_{rel} for each $c_{rel} \in c_{ev}.related\_entities$ together, we get T.

Each c_{rel} is processed sequentially. First we find pairs T_{rel} and then we create TemplLEvents nodes for each $(n, m) \in T_{rel}$, before proceeding to the next c_{rel}. These two steps work as follows:

1. We construct the matcher that finds the (n, m) pairs of T_{rel}. This matcher essentially serves the same purpose as Algorithm 1, which we discussed in Section 4.2. The matcher starts out as the matcher that was used to create TemplLEvent nodes for the input nodes in N, shown on Line 1 of Listing 5.3. This matcher has one variable 'common', which represents all input nodes $n \in N$.

   As an example, take c_{rel} = "<Change_Activity:>Service_Component", which is the *related entity type string* that was used to create the matcher on line 1 of Listing 5.4.

   c_{rel} is split on the colon character, resulting in an array. The resulting array consists of elements that specify a direction ('<'/'>') and a node label. For our example, This array looks like ["<Change_Activity", ">Service_Component"]. The elements of this array are processed sequentially, each adding a part to the starting matcher, according to their direction and label.

   For our example ["<Change_Activity", ">Service_Component"] with $c_e.label$ = 'Change', this goes as follows:

   i start:
   `(common:Change)`

   ii "<Change_Activity":
   `(common:Change)<--(:Change_Activity)`

   iii ">Service_Component":
   `(common:Change)<--(:Change_Activity)-->(related:Service_Component)`

   The final node matcher that is added, in this case the one for Service_Component, is assigned the variable *related*, which finds the input nodes $I_{rel(n)}$ for each $n \in N$. This matcher thus matches all $(n, m) \in T_{rel}$. Note that each (n, m) can be part of the set T_{rel} of multiple related entity strings in $c_{ev}.related\_entities$. If this occurs then that means that m can be reached from n via multiple paths specified in $c_{ev}.related\_entities$.

2. Recall that we want to create one TempLEvent node for each input node pair $(n, m) \in T$. For each $(n, m) \in T_{rel}$ we *merge* a TempLEvent node. To merge a node means to create it if no exact copy of the node already exists. We use the merge operation here, so that we don't create two TempLEvent nodes for one pair (n, m).

   Listing 5.4 shows an example query that creates TempLEvent nodes for BPI 14's 'Change' event schema. The matcher on line 1 of Listing 5.4, was created using the related entity type string equal to '<Change_Activity:>Service_Component'.

   In this query, we first use the matcher we constructed earlier on line 1. This gives us all $(m, n) \in T_{rel}$. Then, for each (m, n), we use the merge operator on line 2. This creates TempLEvent nodes. Let $t_{m,n}$ be the TempLEvent node created to represent (m, n). $t_{m,n}$ is assigned the following two attributes.

   - $t_{m,n}[\text{originID}] \leftarrow ID(m)$
   - $t_{m,n}[\text{commonID}] \leftarrow ID(n)$

   These two attributes thus have to be unique in order for the TempLEvent to be created. If the TempLEvent node is created, we look up the entity object c_e with $c_e.label = m.label$ and set the following the attributes on $t_{m,n}$ on lines 3 and 4 of Listing 5.4.

   - $t_{m,n}[\text{EntityType}] \leftarrow c_e.label$
   - $t_{m,n}[\text{IDraw}] \leftarrow m[c_e.id\_column]$

   We set these two attributes only after creation, because we don't want the merge operation to take these into consideration.

```
1 MATCH (common:Change)<--(:Change_Activity)-->(related:Service_Component)
2 MERGE (n:TempLEvent {originID: ID(related), commonID: ID(common)})
3 ON CREATE SET n.EntityType="Service_Component"
4 ON CREATE SET n.IDraw=related.ID
```

Listing 5.4: Creation of template events of Service_Component nodes related via a Change_Activity node

After performing these steps for every $c_{rel} \in c_{ev}.related\_entities$ we have created a TempLEvent for every $(n, m) \in T$. Earlier, we already created one TempLEvent node per $n \in N$ using the query of Listing 5.3. For each n, we add (n, n) to T, because n is also in $I_rel(n)$ (See Section 4.2). We have now completed the first step in processing entity config c_e and can now continue to the second step, which uses the TempLEvent nodes we just created.

Figure 5.6 shows what the event graph of our example database would look like at this stage. For clarity we don't show the 'NodeID' attribute on nodes other than Entity nodes, but in reality they are there.

Step 2: Creating LEvent and Event nodes

We now create the actual LEvent and Event nodes for N's events. Recall that we have a set T which contains input node pairs (n, m). Every (n, m) is represented by a TempLEvent node. The following properties about n and m hold:

- n is an input node with event(s)
- m is an input node that should relate to the event(s) of n

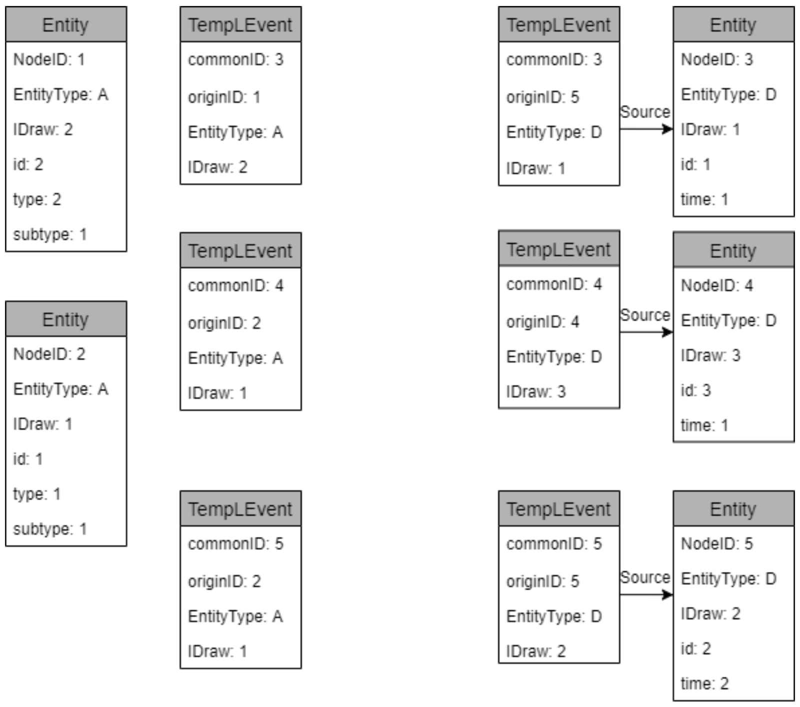

Figure 5.6: Event graph after creating TempLEvent nodes and Source relationships

- $n.label = c_e.label$

Recall that each TempLEvent node representing an input node pair $(n, n), n \in N$ has a Source relationship to n. For each event definition $c_{ed} \in c_e.events.create\_from$, we use the TempLEvent nodes representing T to create LEvent and Event nodes. Listing 5.5 shows an example query of the creation of LEvent and Event nodes for BPI 14. Here we use the event definition c_{ed} for 'Scheduled_Downtime_Start' events of the 'Change_events' event schema, of which the configuration file equivalent can be seen in Listing 5.1.

```
// Find Change TempLEvents that should generate this event
MATCH (temp:TempLEvent {EntityType:'Change'})-->(source)
WHERE 'Scheduled_Downtime_Start' in keys(source)

// Find other matching TempLEvent
MATCH (t:TempLEvent {commonID: temp.commonID})
WITH temp, source, t

// Create LEvent nodes
CREATE (levent:LEvent)
SET levent = t
SET levent.Activity = 'Change: Scheduled_Downtime_Start'
SET levent.Start = source.Scheduled_Downtime_Start
SET levent.End = source.Scheduled_Downtime_Start

WITH temp, collect(levent) as levents

// Create Event nodes
CREATE (ev:Event)

WITH co, levents
UNWIND levents as levent
WITH co, levent

// Create relationships between levents and common nodes
```

```
26 CREATE (levent)-[ec:LE_E {entityType: levent.entityType}]->(ev)
```

Listing 5.5: Creation of Event and LEvent nodes with Scheduled_Downtime_Start as their start time column

The creation of LEvent and Event nodes for an event definition c_{ed} of an entity config c_e works as follows:

1. We find all pairs of TempLEvent nodes that represent each $(n, n) \in T$ and pair them with n itself. We need input node n, because this node contains the event information we need to construct the events as specified by c_{ed}. This works as follows:

   i We find TempLEvent nodes TE of type $c_e.label$. These TempLEvent nodes represent the (n, n) pairs in T. (line 2)

   ii We pair each $te \in TE$, with the input node to which it has a Source relationship. This pairs up each te with their input node n. (line 2)

   iii Each TempLEvent node te whose paired input node n for which $n[c_{ed}.start\_column] = Null$ is removed from TE. These TempLEvent nodes are removed because their respective input node n does not have the event described by c_{ed}. (line 3)

2. We pair each TempLEvent node representing input node pair $(m, n) \in T, m \neq n$ with the TempLEvent node representing input node pair (n, n). We do this because each input node m for which there is a pair $(m, n) \in T$ also relates to the events of n. (lines 6,7).

3. For each TempLEvent te representing an input node pair $(m, n) \in T$ (also those where $m = n$), we create an LEvent node le (line 10). le is thus the LEvent node representing input node pair (m, n). le is assigned a number of attributes. Some are inherited from TempLEvent te and some are inherited from input node n:

   - $le[p] = te[p]$ for any attribute p of te. (line 11)
   - $le[Activity]$ gets a value based on n and $c_{ed}.activity$ as described in Section 5.2.1. (line 12)
   - $le[Start] = n[c_{ed}.start]$ (line 13)
   - $le[End] = n[c_{ed}.end]$ (line 14)

4. For each $n \in N$, we create one Event node e and create an LE_E relationship to e from each LEvent node representing an input node pair (m, n). By doing this, we connect all LEvent nodes of an event to the Event node that represents that event (Lines 19-26).

After performing these steps for one event definition c_{ed}, we have created all LEvent and Event nodes for input node pairs T and event definition c_{ed}. All LEvent nodes also have an LE_E relationship to their corresponding Event node. This is repeated for all other event definitions in $c_e.ES.ED$, after which we have created all events of entities of type $c_e.label$.

We then remove all TempLEvent nodes from the graph and continue with the next entity configuration that specifies event definitions and go back to **step 1**. We repeat this until all entity configuration with event definitions have been handled, which concludes the event creation process.

Figure 5.7 shows what the event graph of our example database would look like at this stage.

5.2.4 Relating entities and events

```
MATCH (ev:LEvent)
MATCH (en:Entity {IDraw: ev.IDraw, EntityType:ev.EntityType})
CREATE (ev)-[r:E_EN]->(en)
SET r.EntityType = en.EntityType
```

Listing 5.6: Creating relationships between Entity and LEvent nodes

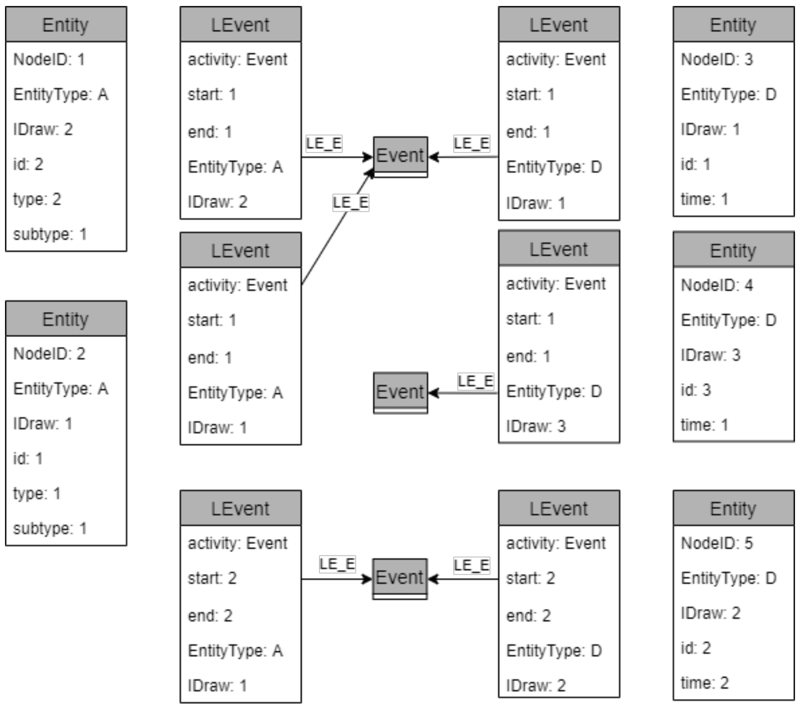

Figure 5.7: Event graph after creating Event and LEvent nodes

Relating Entity nodes with LEvent nodes is quite simple and is shown in Listing 5.6. We simply find, for each LEvent node, the Entity node with the same *IDraw* and *EntityType* and create an E_EN relationship between them.

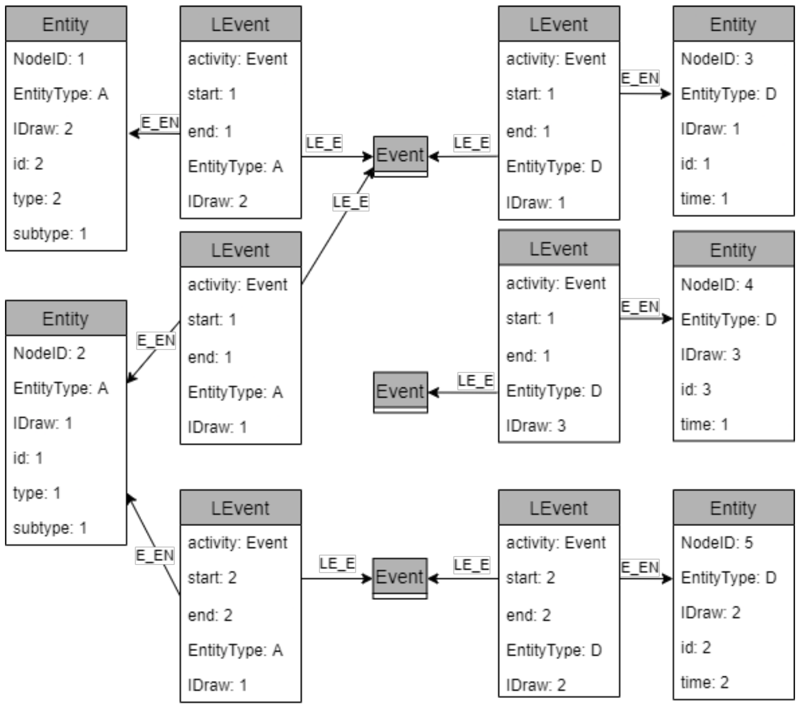

Figure 5.8: Event graph after creating E_EN relationships

5.2.5 Calculating directly follows relations

```
1  MATCH (n:Entity)
2  MATCH (n)-[]-(ev)
3
4  WITH n, ev as nodes ORDER BY ev.Start, ev.commonID
5  WITH n, collect(nodes) as nodeList
6  WITH n, apoc.coll.pairsMin(nodeList) as pairs
7  UNWIND pairs as pair
8  WITH n, pair[0] as first, pair[1] as second
9
10 CREATE (first)-[df:DF]->(second)
```

Listing 5.7: Calculating DF relations between LEvent nodes

After this, we create the 'directly follows' relations, as shown in Listing 5.7. First, we select all Entity nodes and LEvent nodes connected to those Entity nodes, as shown on lines 1-2. Next, we group the LEvent nodes by Entity node and order them by their start time, followed by their *commonID*, as shown on line 4-5. We sort using the *commonID* in addition to the start time, in order to consistently order LEvent nodes with the same start time, as we described in Section 4.2.4. Note that all LEvent nodes connected to the same Event node have the same *commonID*. After this, we create pairs of LEvent nodes, such that we have a list of pairs. Say we have a list of LEvent nodes $[e_0, e_1, \ldots, e_n]$, the list of pairs would then look like $[[e_0, e_1], [e_1, e_2], \ldots [e_{n-1}, e_n]]$. This process is shown on lines 6-8. For each pair we create a DF relationship, which gives us one path of DF relationships per Entity node.

Figure 5.9 shows what the event graph of our example database would look like at this stage.

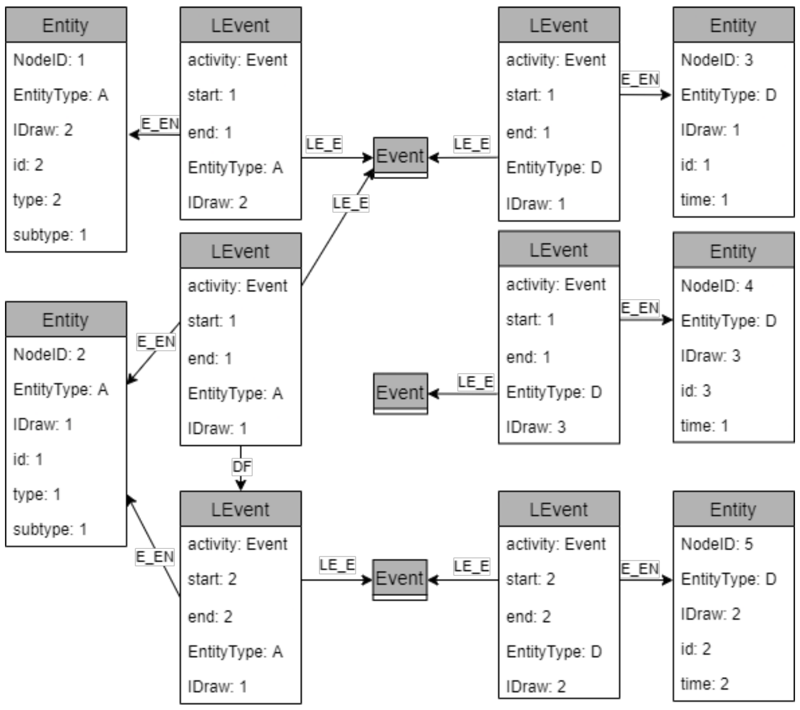

Figure 5.9: Event graph after creating DF relations

5.2.6 Creating the Log node and relating it to events

Finally, we create the Log node and connect it to all LEvent nodes, which is shown in Listing 5.8. This is done by simply creating a single Log node and an L_E relationship between that node and every LEvent node.

Figure 5.10 shows what the full event graph of our example database would look like.

```
CREATE (l:Log {ID: 'BPI14'})
WITH l
Match (e:LEvent)
CREATE (l)-[r:L_E]->(e)
```

Listing 5.8: Creating the Log node and relating it to LEvent nodes

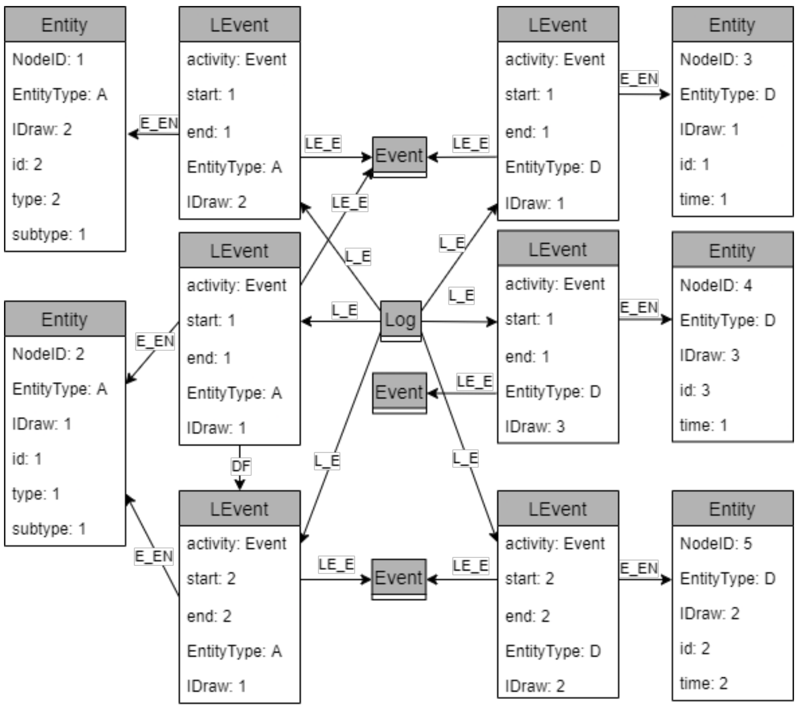

Figure 5.10: Event graph after creating the Log node and relating it to LEvent nodes

Chapter 6

Evaluation

Now that we have proposed our transformation from relational event data to event graph data we need to establish the correctness of this transformation. To do this, we have transformed the normalized versions of BPI 14 and BPI 17 (Sections 2.5, 2.6) to event graphs. The EER diagrams we used for these transformations can be seen in Figures 4.4 and 4.5 respectively. Establishing the correctness of these transformations will mostly rely on a combination of visual inspection and comparing event graph statistics to that of the source relational data. In addition to establishing the correctness of the transformation, we will also discuss some performance statistics of the transformation.

6.1 BPI 14

In this section we will look at the BPI 14 event graph and at some statistics of the BPI 14 data set in the event graph data model so that we can better understand the properties of the event graph and the transformation to the event graph. We also try to argue for the correctness of the BPI 14's event graph.

6.1.1 Event graph exploration

In this section we will take a look at the BPI 14 event graph in order to get a better understanding of it. For this we use the 'Neo4J Desktop' application, which allows you to run Neo4J database instances, as well as query and visualize them. First, take a look at Figure 6.1. Here we see a collection of nodes and relationships. The node type of each of these nodes is indicated by their colour.

- *Pink* nodes represent Entity nodes. Each of these nodes shows its 'EntityType' attribute value.
- *Green* nodes represent LEvent nodes and show its 'Activity' attribute value
- *Light brown* nodes represent Event nodes. These nodes don't show any attribute value

Since there is limited space to show attribute values in these nodes, most attribute values are only partly shown, which is indicated by 3 dots. The top-left Entity node for instance is a 'Change_Activity' Entity node and not a 'Change' Entity node.

Figure 6.1, shows us the trace of the 'Change' entity with IDraw = "C00002243" (bottom-left Entity node). This 'Change' entity has a trace of 4 events. We also show the other LEvent nodes that have an LE_E relationship to events of this 'Change' entity. We do not show the full traces of these other entities, which explains why not all LEvent nodes of the 'Service_Component' entity in the top-right are connected via DF relationships. Listing 6.1 shows the query we used to retrieve all graph components shown in Figure 6.1. We limit ourselves to entities with short traces, because longer traces quickly become hard to read, due to the interconnected nature of graph databases.

```
match (n: Entity {uID:"ChangeBPI14C00002243"}),
(n)<-[een:E_EN]-(le:LEvent),
(le)-[lee:LE_E]->(ev:Event),
(ev)<-[lee2:LE_E]-(le2),
(le2)-[een2:E_EN]->(e2:Entity)
optional match (le)-[df:DF]->()
optional match (le2)-[df2:DF]->()
return *
```

Listing 6.1: Cypher query that finds all graph components shown in Figure 6.1

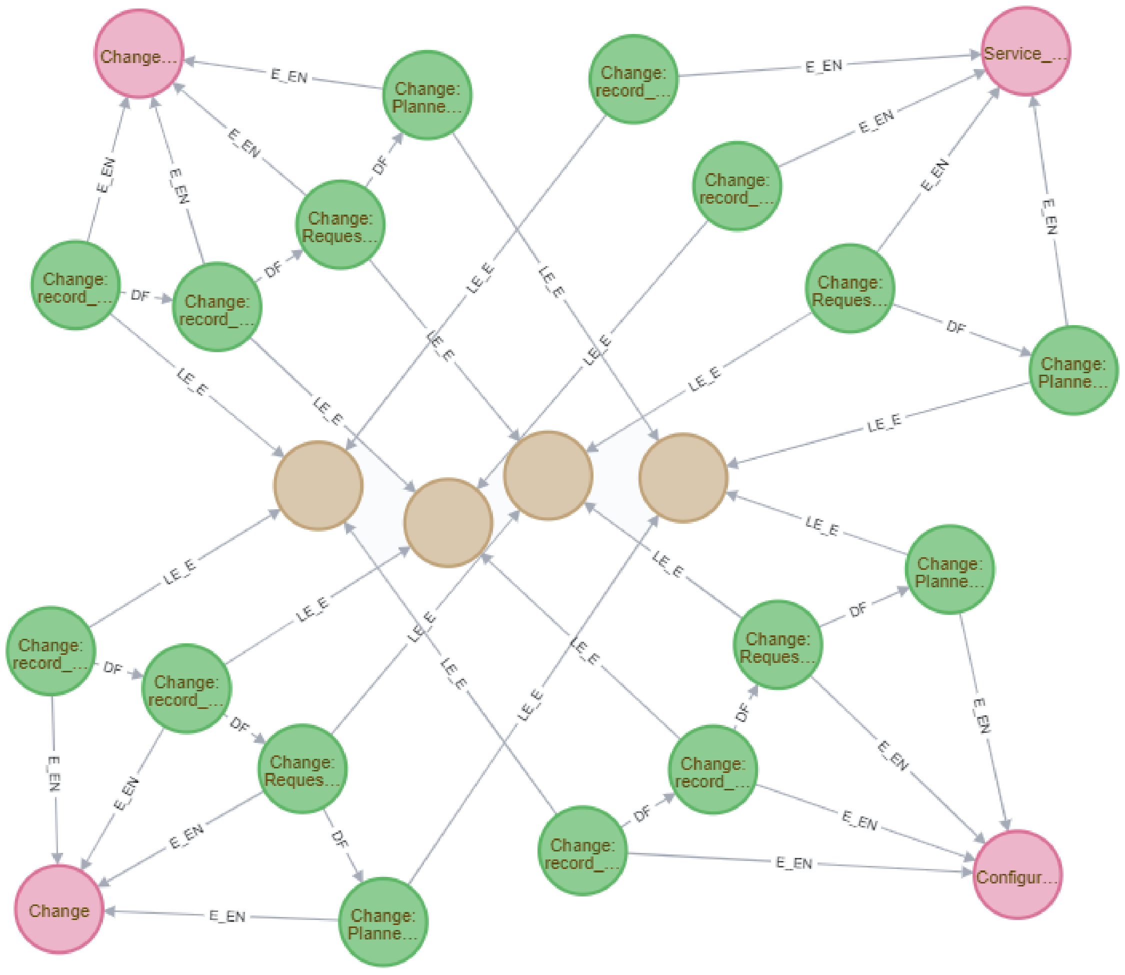

Figure 6.1: Event graph showing the trace of Change 'C00002243' and surrounding entities

Figure 6.2 shows the result of the query in Listing 6.1, but for an 'Incident' entity with IDraw = "IM0035523". The Entity node representing this 'Incident' entity can be seen in the top-middle of the Figure. Again, this 'Incident' entity has a trace of 4 events, but this time there are more other entities and not all of those other entities have all the shown events.

Finally, there is Figure 6.3, which again shows the result of Listing 6.1's query, but for an 'Interaction' entity with IDraw = "SD0000056" (bottom-middle).

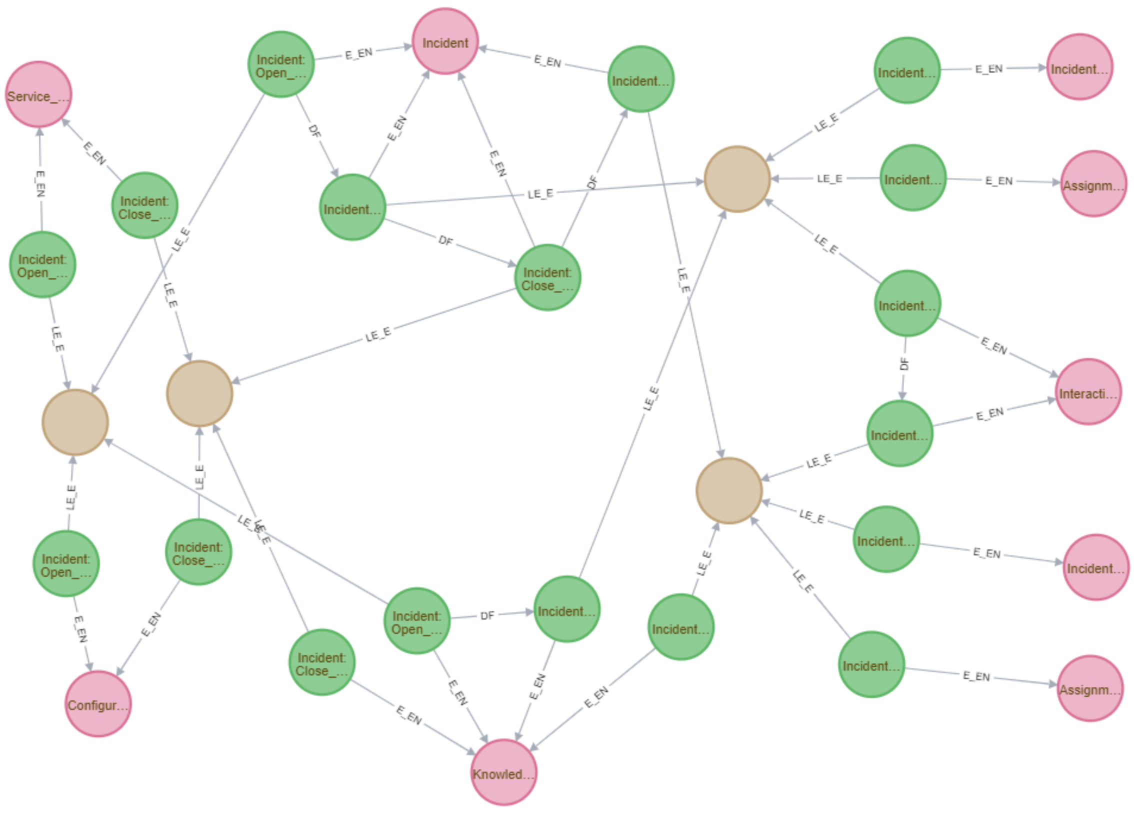

Figure 6.2: Event graph showing the trace of Incident 'IM0035523' and surrounding entities

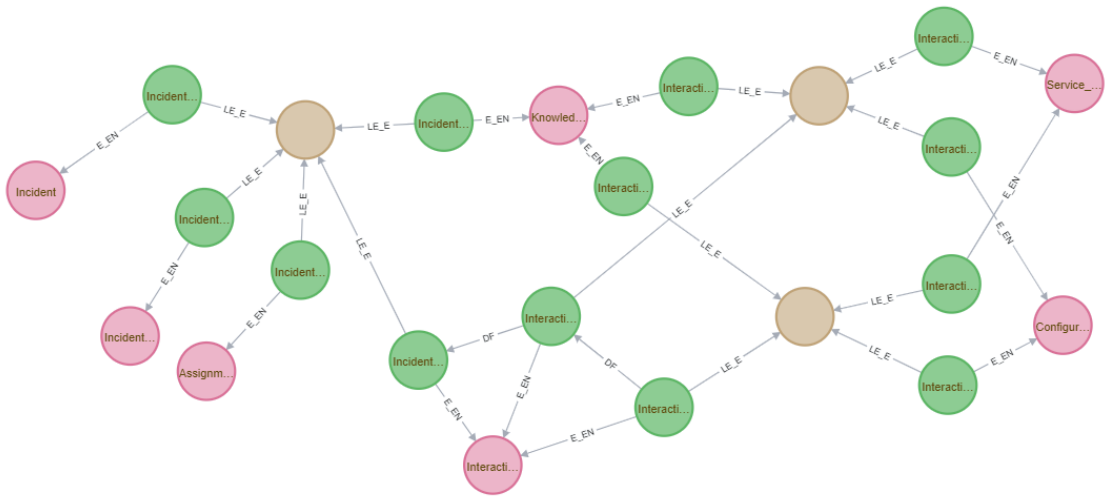

Figure 6.3: Event graph showing the trace of Interaction 'SD0000056' and surrounding entities

6.1.2 Event graph properties

In order to better understand the properties of the event graph, we have gathered several statistics. In this section we will discuss what these statistics tell us about the BPI 14 event graph.

Basic statistics

Table 6.1 shows some basic statistics. What is most note worthy about these statistics is the 'Fill' statistic. This statistic is 0 if the graph has no relationships and 1 if the graph is fully connected. So this statistic tells us the BPI 14 event graph is very sparsely populated with relationships, which is to be expected since every entity relates only to a handful of other entities.

Statistic	**Value**
Volume	18978165 relationships
#Nodes	1834844 nodes
Size	20813009 vertices + relationships
Fill	5.63×10^{-7}relationships/nodes^2

Table 6.1: Basic statistics for the BPI 14 event graph

node type	**average**	**stdev**	**max**	**min**
Event	4.44	0.52	24	3
Entity	6.76	219.58	111222	0
Entity: Incident_Activity	1	0	1	1
Entity: Service_Component	1928.61	7108.51	111222	0
LEvent	1.85	0.35	2	1
Log	0	0	0	0

Table 6.2: Part of in-degree statistics for the BPI 14 event graph shown in Table B.1

node type	**average**	**stdev**	**max**	**min**
Event	0	0	0	0
Entity	0	0	0	0
LEvent	2.85	0.35	3	2
Log	4926161	0	4926161	49261611

Table 6.3: Part of out-degree statistics for the BPI 14 event graph shown in Table B.2

Node degree statistics

Beside these basic statistics, we have also gathered statistics about the in and out degree of various node types. The in-degree of a node is the number of incoming relationships it has. The out-degree of a node is the number of outgoing relationships it has. The in-degree statistics can be seen in Table 6.2 and the out-degree statistics can be seen in Table 6.3.

First, we see that the Event node's in-degree is quite low on average at 4.4, with low standard deviation. However the maximum in-degree is 24, which is quite high considering the low standard deviation. This essentially means that some event relates to 24 entities. In Section 6.1.3 we will look into outliers like these to see whether they correctly represent the relational input data.

Now lets consider the in-degree statistics of Entity nodes, which can strongly vary depending on the data set. These statistics strongly depend on the data set as some entities like service components, are involved in many events, while others like incident activities, are involved in very little events. For this reason, this statistic cannot be used to verify graph correctness.

Then we have LEvent nodes, which have a quite low and consistent in-degree. The in-degree of LEvent nodes can be either one or two, which is reflected in the maximum and minimum in-degree of these nodes. One in-relationship originates from the Log node and the other in-relationship is present if another LEvent node has a DF relationship to it. This statistic shows how many LEvent nodes are the first LEvent node in their DF-path divided by the total number of LEvent nodes. If this statistic is one, then no LEvent nodes have an incoming DF relationship, i.e. there are no DF relationships. If this statistic is two, then either there are no LEvent nodes, or all LEvent nodes have an incoming DF relationship, which means there exist cycles of DF paths, which is not allowed.

Now consider Table 6.3, which shows statistics about the out-degree of LEvent nodes instead. The LEvent out-degree statistics tell the same story as their in-degree statistics. Every LEvent node has an outgoing relationship towards an Entity and Event node. The remaining relationship describes the ratio of first LEvent nodes like the in-degree described the final LEvent nodes. Unsurprisingly these ratios are the same, which leads us to believe LEvent nodes are compliant with our event graph data model of Figure 3.8.

Finally there is the Log node, whose degree is simply equal to the number of LEvent nodes in the graph.

Node degree histograms

In addition to these degree statistics, we have also constructed some histograms, describing the degree of nodes. Note that the Y-axis of these histograms is on a logarithmic scale. If we look at Figure 6.4, we see that Event nodes with a low in-degree between 3 and 7 are by far the most common, while there are very few Event nodes with a higher in-degree.

The distribution of the Event nodes' in-degree heavily depends on the data set. In BPI14's case, these higher in-degrees exclusively occur for events created from the Change table, which is why there is such a steep drop around an in-degree of 7. The Change table is also the only table which is related to other entities via a transitive entity. While this doesn't have to result in high in-degrees, it does lend itself to higher in-degrees, since a larger part of the input graph is explored to find related entities, with respect to not using a transitive entity.

Then, if we look at Figures 6.5-6.7, we see that Entity nodes with little events are the most common, and the number of occurrences of Entity nodes with more events decreases, the more events are related to those Entity nodes. Even though the shape of the Incident Entity and Knowledge_Document Entity histograms are quite similar, their x-axes are quite different. This is also true for the histograms of other entity types. From this we gather that the rate at which the number of occurrences declines is somewhat proportional to the difference between the smallest and largest in-degree for that entity type.

Note that this isn't true for Entity nodes overall, as can be seen in Figure 6.5, since this mixes the in-degrees of several entity types. However, it is still clear that a lower in-degree is more common.

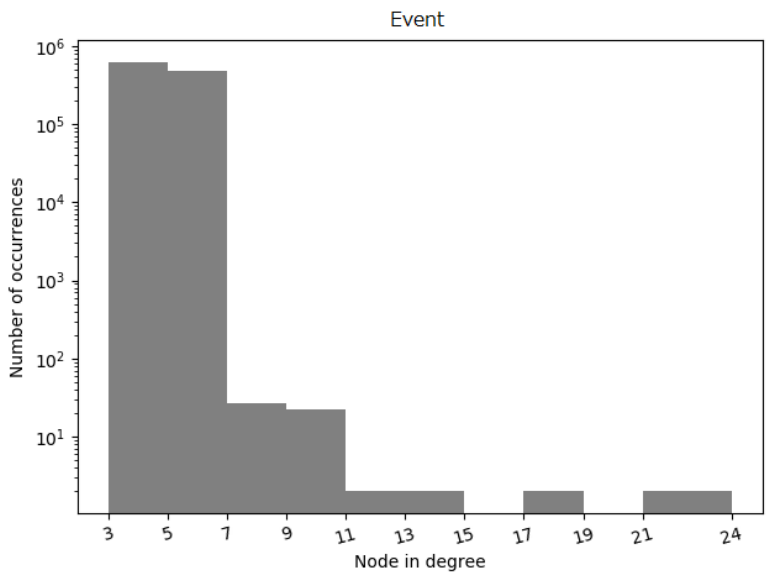

Figure 6.4: Histogram of Event nodes' in degree (BPI14)

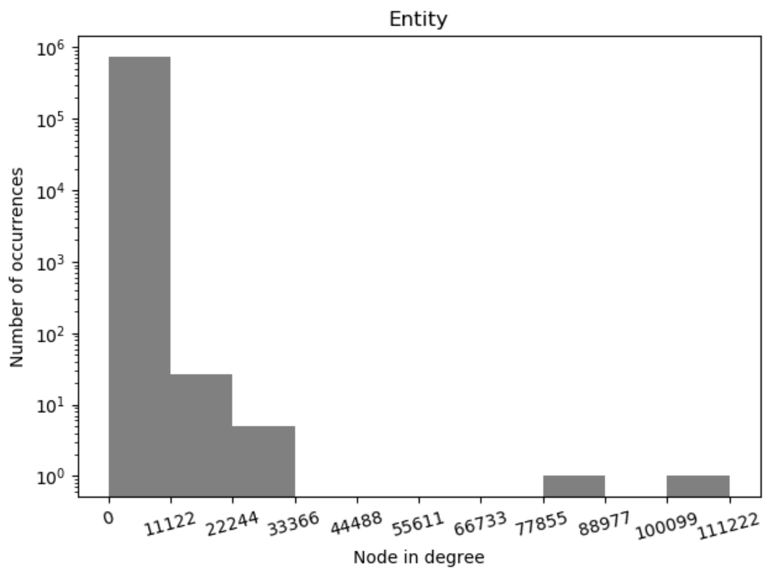

Figure 6.5: Histogram of Entity nodes' in degree (BPI14)

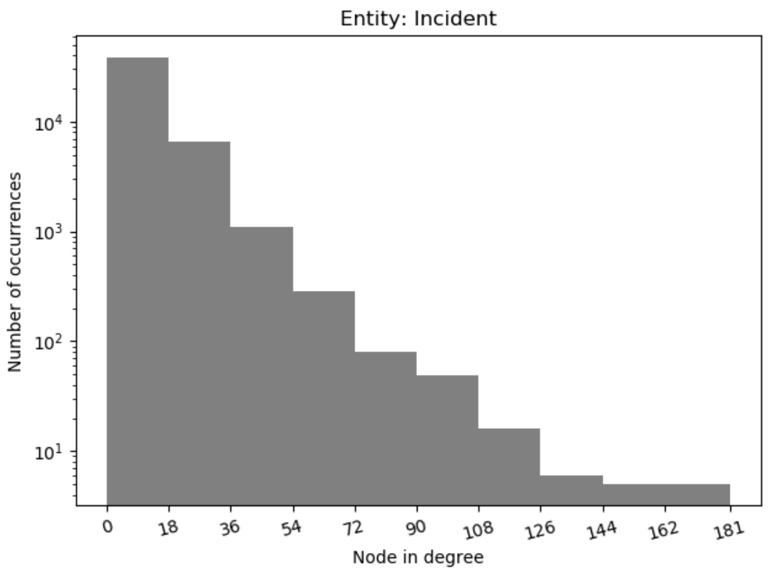

Figure 6.6: Histogram of Incident Entity nodes' in degree (BPI14)

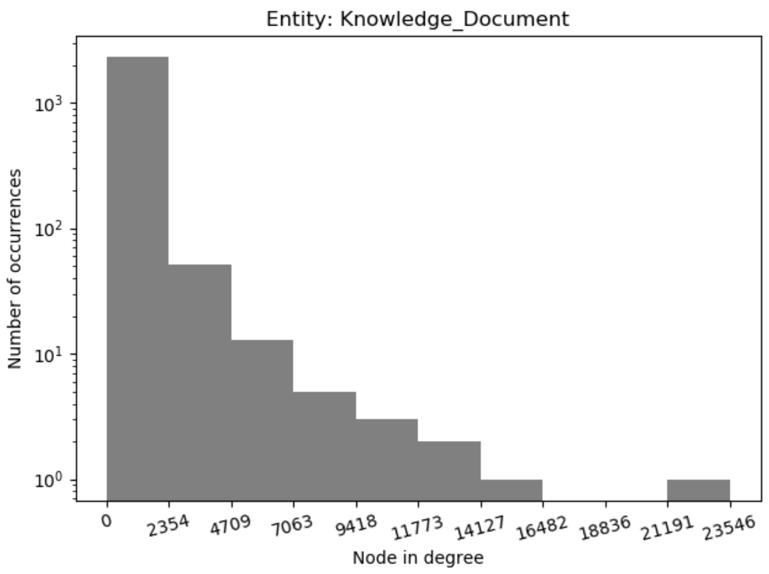

Figure 6.7: Histogram of Knowledge_Document Entity nodes' in degree (BPI14)

6.1.3 Event graph correctness

It is not easy to prove that the BPI 14 data set has been correctly transformed to an event graph. However, we can create some confidence that this is the case. In this Section we verify the correctness of as many event graph components we could. What we cannot verify, we will leave to visual inspection of the BPI 14 event graph.

Verifying the correctness of Entity nodes

To verify whether each entity is represented in the event graph, we compare the number of occurrences of each entity type in the source data with the number of occurrences of each entity type in the event graph. This is shown in Table 6.4. From this table we conclude that all entities are represented in the event graph. We cannot easily verify whether each entity has the correct attributes, as the number of attributes can vary per entity, depending on their type and whether they have any *null* values, and verifying whether every attribute value is correct is very computationally expensive. However, this looks to be the case from visual inspection of the event graph.

	Number of occurrences	
Entity type	**source data**	**event graph**
Configuration_Item	15864	15864
Change	18000	18000
Change_Activity	30275	30275
Incident	46809	46809
Incident_Activity	466737	466737
Interaction	147004	147004
Knowledge_Document	2374	2374
Service_Component	342	342

Table 6.4: Occurrences of entity types in the source data versus the BPI 14 event graph

Verifying the correctness of events

We verify whether all events are represented in the event graph in a similar way. Recall that every event was creating by using an event definition. Each event definition belongs to a relational database table and specifies a start column, which is used to set the event's start time. Therefore, to check whether every event in the relational data is represented in the event graph, we calculate how many rows in the relational database have a non-*null* value in the start column specified by each event definition. Each of those rows should have resulted in one event (per relevant event definition) during the transformation. Each of these row counts should thus coincide with the number of Event nodes in the graph that were created by using those same event definitions. These counts can be seen in Table 6.5, where we see that this is indeed true.

We can not easily verify whether every LEvent node in the event graph is correct. Therefore we rely on the visual inspection of the event graph for this.

Verifying the correctness of the Log node

To verify the correctness of the Log node, we simply check if it exists in the graph and whether it's 'ID' attribute is equal to 'BPI14', which it is. Therefore we conclude that BPI 14's Log node is correct.

Verifying the correctness of E_EN relationships

To verify the correctness of E_EN relationships, we verify the following properties:

Table	Start_Column	Number of occurences source data	event graph
Change_Activity	Actual_Start	27017	27017
	Actual_End	27014	27014
	Planned_Start	30275	30275
	Planned_End	30232	30232
	Requested_End_Date	30275	30275
	Change_record_Open_Time	30275	30275
	Change_record_Close_Time	30275	30275
Incident_Activity	DateStamp	466737	466737
Change	Scheduled_Downtime_Start	384	384
	Scheduled_Downtime_End	382	382
Incident	Open_Time	46606	46606
	Reopen_Time	2284	2284
	Resolved_Time	44826	44826
	Close_Time	46606	46606
Interaction	Open_Time_First_Touch	147004	147004
	Close_Time	147004	147004

Table 6.5: Occurrences of events per 'event definition start column' in the source data versus the BPI 14 event graph

1. Every E_EN relationship's source is an LEvent node
2. Every E_EN relationship's target is an Entity node
3. the LEvent and Entity nodes which each E_EN node connects, have the same 'IDraw' attribute.

These properties all hold, which means that every E_EN relationship is correct.

Verifying the correctness of LE_E relationships

We cannot easily verify whether each Event node has LE_E relationships with the correct number of LEvent nodes, so we will rely on the visual inspection of the BPI 14 event graph to see whether we can find any Event nodes for which this is not true.

Verifying the correctness of DF relationships

We require that the start time of the source node of a DF relationship is either earlier or the same as the target node of a DF relationship. This holds for every DF relationship in the event graph. Furthermore, as stated in Section 4.2.4, we require that in case two events have the exact same start time, the DF relationship ordering has to remain consistent among LEvent nodes connected to the same Event nodes. To verify whether this is true in the BPI 14 event graph, we queried (Listing 6.2) the event graph using a pattern that matches DF relationship for which this does not hold and found none of such cases.

```
match (e1a:LEvent)-->(c1:Event)<--(e1b:LEvent),
    (e1a)-[df1:DF]->(e2a:LEvent)-->(c2:Event),
    (e1b)<-[df2:DF]-(e2b:LEvent)-->(c2:Event)
return count(*)
```

Listing 6.2: Cypher query that finds incorrect DF relationship ordering of events with the same timestamp

We also verified for each Entity node, whether the LEvent nodes connected to that Entity node have been connected by a single DF relationship path. To verify this we again queried (Listing

6.3) the graph database to look for Entity nodes for which does not hold. We found no such Entity nodes.

```
match(e: Entity)<--(le: LEvent)
with e, size((le)--()) as degree
where degree < 4
with e, count(degree) as c
where c > 2
return count(e)
```

Listing 6.3: Cypher query that finds entities with disconnected DF paths

Verifying the correctness of the L_E relationships

Checking the correctness of the L_E relationships is simple. We look at the max out-degree of the Log node in Table 6.3, as there is only one Log node in the event graph. This number is consistent with the number of LEvent nodes in the event graph, which means that there is an L_E relationship to every LEvent node, which is all we need for L_E relationships to be correct.

Verifying the correctness of outliers

In Section 6.1.2 we identified outliers in the BPI 14 event graph statistics. In this section we will look at those outliers to verify whether they correctly represent the relational input data.

First, look at Tables 6.2, which shows statistics about the in-degree of various node types. We found that the maximum Event nodes was rather high when compared to its average and standard deviation.

To verify whether this is correct, or the result of a problem with the transformation, we compared the relational source data and event graph data. We found that there was a single Event node with this in-degree, which was constructed using a row from the Change table. In the source relational data, this row is related to 22 Change_Activity rows, which in turn relate to one Service_Component row and 22 Configuration_Item rows. Recall Figure 4.4, which shows Change_Activity is a transitive entity for events created from the Change table. So, if we count the related configuration items, service components and the Change tuple itself, we indeed come to a total of 24 related entities. Thus the in-degree of this Event node is correct and not a transformation error.

In Section 6.1.2 we have shown just a selection of in-degree histograms, but in reality we have constructed one for every node and entity type. A number of these histograms contained outliers, like we see in Figure 6.4. For each of these outliers, we verified whether that outlier is consistent with what we find in the source data, which was the case.

6.1.4 Transformation Performance Statistics

We have transformed the BPI 14 data set into an event graph using the described automatic transformation. An SQLite database, containing only the BPI 14 data, after normalization, costs 692 MB to store. It contains approximately 720 000 rows. The Neo4J database instance that contains the transformed BPI 14 data costs 2.70 GB to store. It contains approximately 6 800 000 nodes and 19 000 000 relationships. There are a couple of reasons for this increase in storage size.

All the input data (except relations) can be found on just the Entity nodes. But in addition to Entity nodes, the graph contains a variety of other nodes, which need to be stored as well. Each LEvent node stores an activity name, start, and end time. LEvent nodes connected to the same Event nodes share the same attribute values. This data could also be stored on the Event node to eliminate data duplication, but doing this makes it harder to analyse, so we've chosen not to do this. There's also the relationships, which need to be stored. In a relational database this is stored in a schema, which hardly costs any storage. However we cannot do this in a graph database and have to store each relationship. Finally, there is some data duplication in the way that Neo4J stores graphs. Since Neo4j stores nodes, attributes and relationships are each stored in separate files. Since properties are stored in a separate file, this file also needs to contain some data that pairs the attributes to the correct nodes and relationships.

The transformation was run on a system with 24 GB RAM (2133 MHz) and a quad core processor (2.6 GHz) without hyper threading. The Neo4J database instance was allocated 20 GB of RAM, as Neo4J recommends to leave 4 GB for the system itself. It took one hour and twenty six minutes to perform the transformation. The time of the transformation is heavily dependent on the size of the input data and whether the system has enough RAM. This is because the performance of Neo4J's query engine dramatically slows down when it does not have enough RAM, and the amount of required RAM depends on the size of the input database.

6.2 BPI 17

In this section we will look at the BPI 17 event graph and at some statistics of the BPI 17 data set in the event graph data model, like we did with BPI 14. We verified the correctness of correctness of the BPI 17 event graph in the same way as we did for the BPI 14 event graph. Since the BPI 17 results are so similar to those of BPI 14, we keep its discussion short.

6.2.1 Event graph exploration

In this section we take a look at the BPI 17 event graph. The Figures we show in this section have been constructed using the query of Listing 6.1 and can be interpreted like the Figures of Section 6.1.1.

Figure 6.8 shows the trace of the application Entity node with IDraw = “Application_228161231”. This application Entity node can be seen on the right side of the Figure. Any other Entity nodes that show ‘applicati...’ represent ‘application_events’ entities. Figure 6.9 shows the trace of the offer Entity node with IDraw = “Offer_1191705426” (right side) and Figure 6.10 shows the trace of the resource Entity node with IDraw = “User_145” (middle).

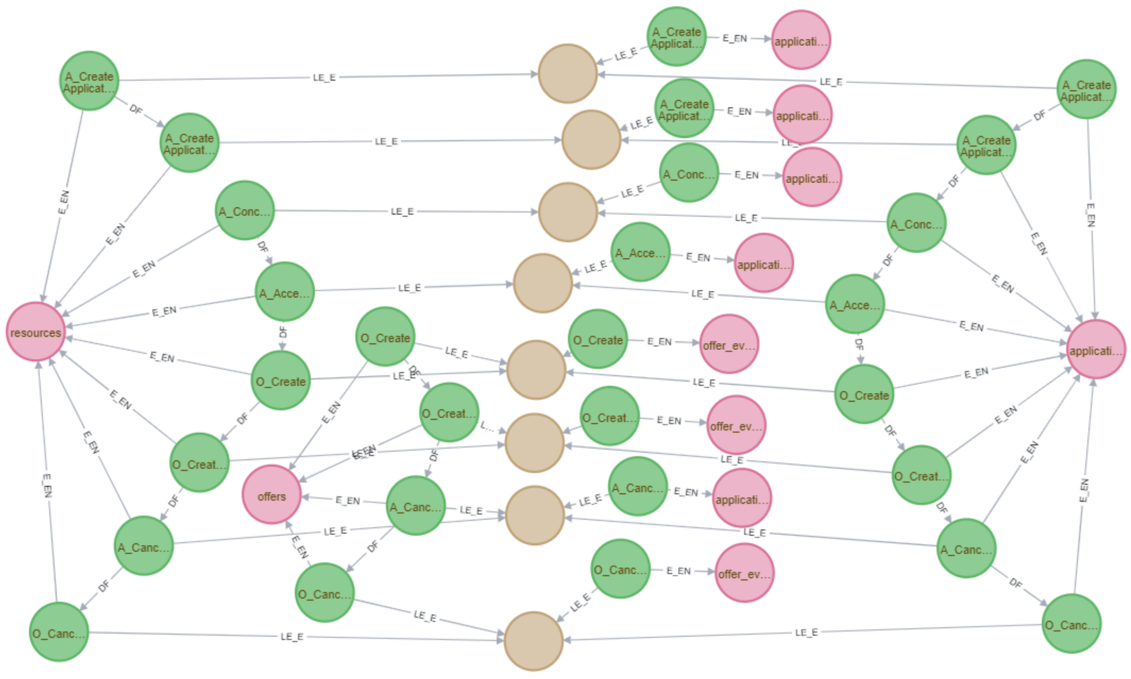

Figure 6.8: Event graph showing the trace of application 'Application_228161231' and surrounding entities

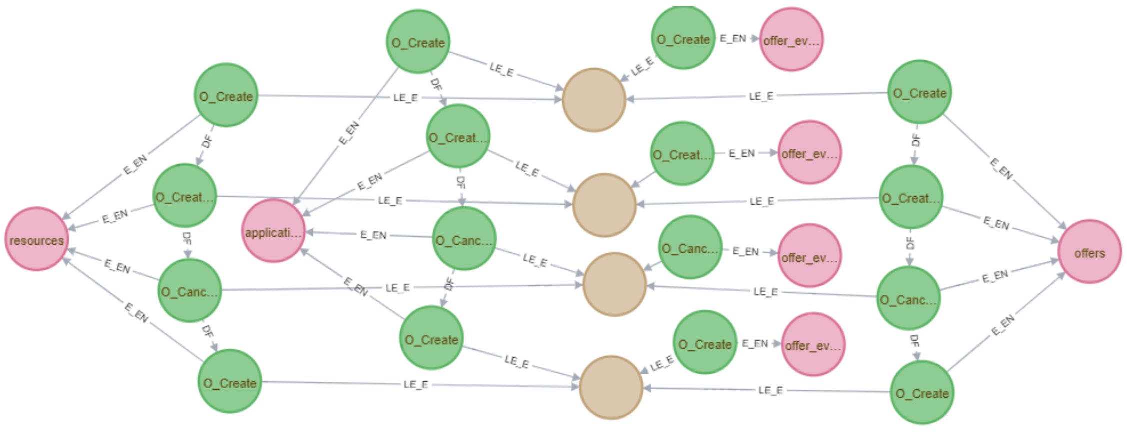

Figure 6.9: Event graph showing the trace of offer 'Offer_1191705426' and surrounding entities

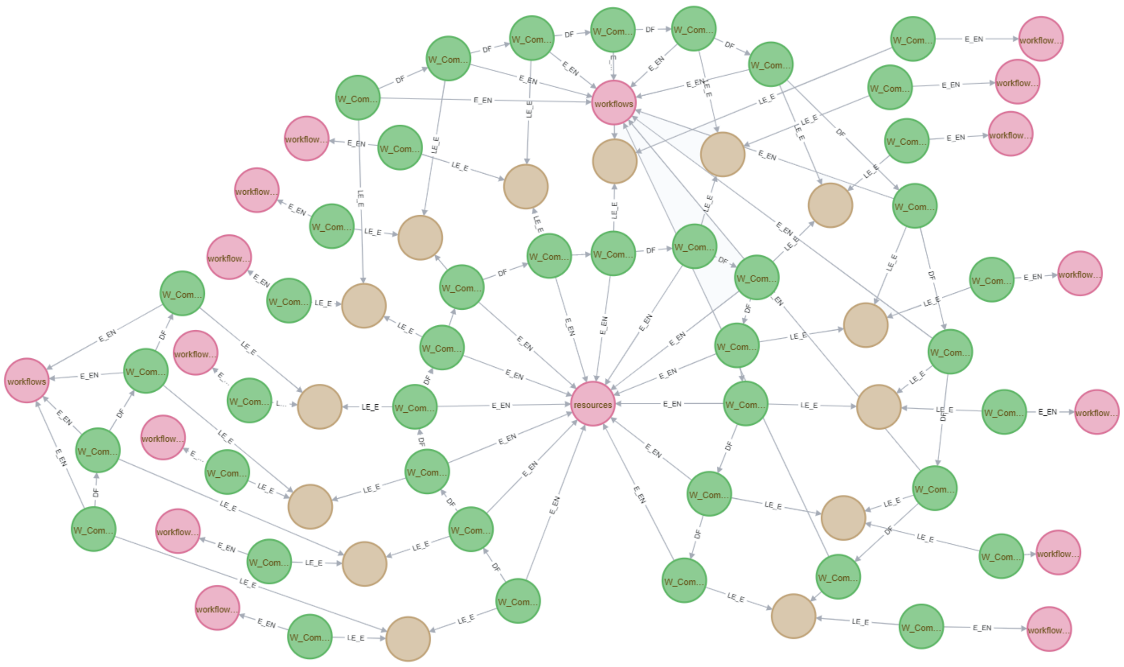

Figure 6.10: Event graph showing the trace of resource ‘User_145’ and surrounding entities

6.2.2 Event graph properties

We have gathered the same set of statistics about the BPI 17 data set as we did about the BPI 14 data set. We see similar results between the statistics of the two data sets.

Basic statistics

In Table 6.6 we listed some basic statistics. We see that the *Fill* statistics is a bit higher than that of of BPI 14. This is because the EER diagram we made for BPI 17 (Figure 4.5) is more connected then that of BPI 14 (Figure 4.4).

Statistic	Value
Volume	16600218 relationships
#Nodes	7527121 nodes
Size	19660646 vertices + relationships
Fill	1.77×10^{-6}relationships/nodes^2

Table 6.6: Basic statistics for the BPI 17 event graph

Node degree statistics and histogram

Take a look at Tables 6.7 and 6.8. The statistics we see here are very similar to those of the BPI 14 event graph, which can be seen in Tables 6.2 and 6.3. Again, there is a large difference between the in-degree statistics of different entities, which is expected. The maximum in-degree of Entity nodes, is very high (138130). we found this to be an outlier of the 'resources' entity type, which can be seen in Figure 6.11. We found that this resource was 'User_1'. We assume that this resource has some sort of special function in the process described by BPI 17. We also verified the correctness of any other outliers we found in the 'degree' histograms we constructed for BPI 17.

If we look at the minimum and maximum in and out-degree of LEvent nodes, we see that they are consistent with what we saw in Tables 6.2 and 6.3 for BPI 14. The out-degree of the Log node is also consistent with the number of LEvent nodes again.

node type	average	stdev	max	min
Event	3.85	0.55	5	3
Entity	2.35	128.8	138130	0
Entity: applications	34.44	15.91	176	8
Entity: resources	8002.79	12418.40	138130	14
LEvent	1.72	0.45	2	1
Log	0	0	0	0

Table 6.7: Part of in-degree statistics for the BPI 17 event graph shown in Table C.1

node type	average	stdev	max	min
Event	0	0	0	0
Entity	0	0	0	0
LEvent	2.72	0.45	3	2
Log	4466693	0	4466693	4466693

Table 6.8: Part of out-degree statistics for the BPI 17 event graph shown in Table C.2

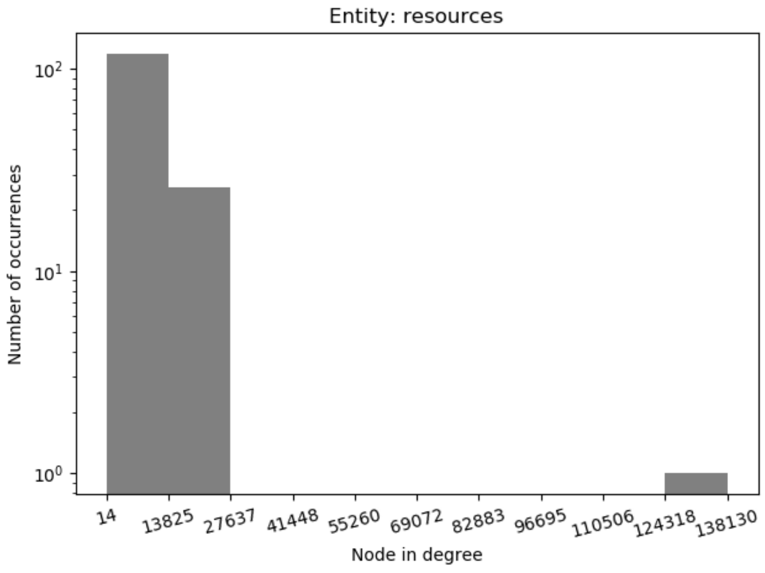

Figure 6.11: Histogram of resources Entity nodes' in degree (BPI17)

6.2.3 Event graph correctness

We performed the same verification steps for BPI 17 as we did for BPI 14 and found no errors. Table 6.9, shows the comparison of entity counts between the source data and the event graph. Table 6.10 shows the comparison of event counts between the source data and the event graph.

In Section 6.2.2 we found that the resource 'User_1' had a much higher in-degree than other resources. We verified whether this user was actually involved in this many events by looking at the original (denormalized) *.csv* file and found the in-degree of this Entity node to be consistent with the number of rows in the *.csv* file in which 'User_1'' was the resource. We verified the correctness of other outliers in a similar way.

Since we found no errors in the event graph, we have confidence that the BPI 17 data set has been correctly transformed to an event graph.

	Number of occurrences	
Entity type	**source data**	**event graph**
offer_events	352149	352149
offers	42995	42995
application_events	633468	633468
applications	31509	31509
workflow_events	174788	174788
workflows	31500	31500
resource	145	145

Table 6.9: Occurrences of entity types in the source data versus the BPI 17 event graph

6.2.4 Transformation Performance Statistics

Like with BPI 14, we transformed the BPI 17 data set into an event graph using the described automatic transformation. An SQLite database, containing only the BPI 17 data, after normalization, costs 1.0 GB to store. It contains approximately 1 267 000 rows. The Neo4J database instance that contains the transformed BPI 17 data costs 2.95 GB to store. It contains approx-

		Number of occurences	
Table	**Start_Column**	**source data**	**event graph**
application_events	startTime	633468	633468
offer_events	startTime	352149	352149
workflow_events	startTime	174788	174788

Table 6.10: Occurrences of events per 'event definition start column' in the source data versus the BPI 17 event graph

imately 7 500 000 nodes and 16 600 000 relationships. The transformation was run on the same system as the BPI 14 transformation and took two hours and ten minutes to complete.

Chapter 7

Conclusions

In this Thesis, we discussed how we can automatically transform event data from a relational to a graph database, where we have one-to-one, one-to-many and many-to-many relationships between events and case identifiers such that the transformed data captures both structural and temporal relations in the data.

Esser proposed a data model to store such data in a graph database. Due to us using data from a relational database as input to our transformation, instead of Esser using event logs as input to his transformation, we had to make some adjustments to his proposed data model. In essence this means that each event is represented by multiple nodes, instead of just one like Esser proposed. This also entailed adjusting how behaviour of compound entities in labelled property graphs can be represented, which now requires adding both nodes and relationships to the existing event data grpah, instead of just relationships.

Not all relational databases are suitable for the transformation we designed, so we had to impose some restrictions on the input data, for it to be used. Some of these restrictions limit the range of names allowed to be present in the relational schema, but the main restriction is that each table should contain exactly one entity per row and each row of the same table should contain the same type of entity.

By making this ‘one entity per row’ assumption, we could store both event and entity attributes on entity nodes, instead of storing event attributes on event nodes. This allowed us to circumvent the issue of storing multiple, equally named, event attributes on one Event node and also allowed us to prevent data duplication of these event attributes.

Transforming a relational database to an event data graph is done in multiple steps. The first step is to ensure the input data conforms to our imposed restrictions, after which we use R2PG-DM to transform the relational database into a generic property graph. The output of R2PG-DM then has to be transformed such that we can import it using Neo4J’s ‘admin import’ tool.

At this point we require some domain knowledge to continue with the transformation, in the form of an EER diagram. This EER diagram describes where entities and events can be found in the input data and how events should relate to entities.

After such an EER diagram has been constructed, the transformation from the generic property graph, the event data graph can begin. First, Entity nodes are created, after which we create Event and LEvent nodes. These Event and LEvent nodes are related using LE_E relationships, so that we know which LEvent nodes belong together. At this point we add E_EN relationships between each LEvent nodes to an Entity node and add DF relationships between LEvents. Finally, single Log node is created and related to all LEvent nodes via L_E relationships.

To verify that the transformation works, we applied it to the BPI 14 and BPI 17 data sets. We found that quite a large amount of RAM is needed to complete the transformation in a reasonable time. We also found that it costs quite a bit more storage space, to store the event data graph, when compared to storing the relation database. This is mostly due to the event structures, which are not present in the relational database and the fact that relationships need to be explicitly stored, instead of implicitly via the schema of a relational database. We also looked at outliers in statistics on this event graph data, to make sure these were not errors in the transformation. We found these outliers to be correctly represented and from extensive general inspection of the event graph data, we concluded that, as far as we could tell, both the BPI 14 and BPI 17 data set have been transformed successfully.

7.1 Limitations and future work

Due to the design of the transformation, there are several restrictions that a relational database has to conform to in order to be used in the transformation. Firstly, there are some constraints on the names tables can have. Future work could look into changing the transformation, such that these constraints can be lifted.

The first of these constraints is that table names cannot contain spaces. This is not a problem for the transformation discussed in this thesis, but instead for R2PG-DM, which is used in the transformation pipeline. Theoretically this is not an issue. However the tool's implementation does not support this.

The second constraint on table names is that they cannot contain any of the characters '<', '>' or ':'. These characters are not allowed because they are special characters used to configure the transformation, as described in Section 5.2.1. This constraint could be removed by allowing these characters to be escaped in the configuration file.

Finally, the third constraint is that it is not allowed to have a table with the name 'edge', 'node', 'property', or any of the node types created during the transformation. These tables are not allowed to exist in the input database, because they are assumed to not exist and breaking this assumption would lead to a faulty transformation and potentially loss of data.

Another limitation is that each table's primary key can only consist of one column. This is due to the way we have implemented our transformation, but we believe the transformation can easily be adjusted to allow for multi-column primary keys. Alternatively, a new single column primary key can always be added to an existing relational database to get around this limitation.

The last limitation, as discussed in Section 3.2.6, is that Entity nodes are constructed using a single row in the input database. This has some implications on the granularity of the data. Consider a table where each row describes an entity. If this table contains some attribute that changes over time, e.g. 'total-cost', then this attribute is added to the Entity nodes that represent entities from this table. However, on these LEvent nodes, there is nothing that distinguishes this attribute from static attributes, which don't change over time. This means that without domain knowledge, you cannot immediately see which attributes are static or dynamic and could interpret the data in the wrong way. Future work could look into an alternative solution to storing event attributes, so that this domain knowledge is no longer required to fully understand the data.

In our transformation, we did not include the creation of HOW relationships between Entity nodes. We left these out of the transformation as we were only interested in the structural and temporal representation of event data. Future work could thus look into how these HOW relationships could be added to the event data graph transformation.

Finally, this thesis only looked into the representation of event data in a graph database and the transformation to that representation. We did not explicitly discuss how process mining algorithms should use this data and how this affects the analysis of a process model created by a process mining algorithm using event graph data. Future work could research how this data is best accessed by a process mining algorithm by, for instance, designing an interface to extract data from the event data graph. Future work could also look into new process mining algorithms that are designed to use event graph data. And Finally, future work could discuss the analysis of process models created by such new process mining algorithms.

Bibliography

[1] Neo4J. `https://neo4j.com/`. Accessed: 06/01/2020. 6

[2] OpenCypher. `http://www.opencypher.org/`. Accessed: 06/01/2020. 7

[3] Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework). Standard ISO/IEC 9075-1:2016, International Organization for Standardization, 12 2016. 6

[4] Wil Aalst, Boudewijn Dongen, Christian Günther, Anne Rozinat, Eric Verbeek, and A. Weijters. Prom: The process mining toolkit. 01 2009. 5

[5] Esser, S. A Schema Framework for Graph Event Data. Master's thesis, Technishe Universiteit Eindhoven, 2 2020. 1, 2, 17

[6] Esser, S., Fahland, D. *Storing and Querying Multi-dimensional Process Event Logs Using Graph Databases*, pages 632–644. Springer International Publishing, 2019. 1

[7] Xixi Lu, Marijn Nagelkerke, Dennis Wiel, and Dirk Fahland. Discovering interacting artifacts from erp systems. *IEEE Transactions on Services Computing*, 8:1–1, 11 2015. 1, 9

[8] G. Sanders and Seung Shin. Denormalization effects on performance of rdbms. volume 3, 01 2001. `doi:10.1109/HICSS.2001.926306`. 5

[9] Radu Stoica, George Fletcher, and Juan F. Sequeda. On directly mapping relational databases to property graphs. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay*, 2019. 2, 7

[10] Stoica, R. R2PG-DM: A direct mapping from relational dtaabases to property graphs. Master's thesis, Technishe Universiteit Eindhoven, 7 2019. 2, 7

[11] van der Aalst, W. *Process Mining: Data Science in Action.* Springer, Second edition, 2016. 1, 4, 5

[12] van Dongen, B.F. BPI Challenge 2014. `http://dx.doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35`. 9

[13] van Dongen, B.F. BPI Challenge 2017. `http://dx.doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b`. 12

Appendix A

Example relationship creation of R2PG-DM

```
WITH joinableColumns AS (
  SELECT DISTINCT sourceTable.btype, sourceTable.bsubtype
  FROM "A" as sourceTable
  INNER JOIN "B" as targetTable
    ON sourceTable.btype = targetTable.type
    AND sourceTable.bsubtype = targetTable.subtype
},
sourceNodes AS (
  SELECT n.id AS id, p,pkey AS pkey, p.pvalue AS pvalue
  FROM node AS n
  INNER JOIN property AS p
  ON n.id = p.id
  AND (p.pkey ='type' OR p,pkey = 'subtype')
  AND n.label = "A")
),
targetNodes AS (
  SELECT n.id AS id, p,pkey AS pkey, p.pvalue AS pvalue
  FROM node AS n
  INNER JOIN property AS p
    ON n.id = p.id
    AND (p.pkey ='btype' OR p,pkey = 'bsubtype')
    AND n.label = "B")
),
pivotedSourceNodes AS (
  SELECT id as sourceID,
    MAX(CASE WHEN pkey ='btype' THEN pvalue END) AS btype,
    MAX(CASE WHEN pkey ='bsubtype' THEN pvalue END) AS bsubtype
  FROM sourceNodes AS s
  GROUP BY s.id
),
pivotedTargetNodes AS (
  SELECT id as targetID,
  MAX(CASE WHEN pkey ='type' THEN pvalue END) AS type,
  MAX(CASE WHEN pkey ='subtype' THEN pvalue END) AS subtype
  FROM targetNodes AS t
  GROUP BY t.id
),
joinedSourceNodes AS (
  SELECT s.sourceID, s.type, s.subtype
  FROM pivotedSourceNodes AS s
  INNER JOIN jonableColumns AS j
    ON s.btype = j.btype
    AND s.bsubtype = j.bsubtype
)

```

```sql
SELECT s.sourceID, t.targetID
FROM joinedSourceNodes AS s
LEFT JOIN pivotedTargetNodes AS t
  ON s.btype = t.type
  AND s.bsubtype = t.bsubtype
```

Listing A.1: Example query of how R2PG-DM creates relationships between nodes of two relations

Appendix B

BPI 14

B.1 BPI 14 configuration file

```
{
  "connection": {
    "neo4j": {
      "jdbc": "bolt://127.0.0.1:7687",
      "user": "neo4j",
      "password": "1234"
    }
  },
  "log": {
    "name": "BPI14"
  },
  "entity": [
    {
      "label": "Change_Activity",
      "id_column": "ID",
      "event": {
        "related_entities": ["Service_Component", "
            Configuration_Item", "Change"],
        "create_from": [
          {
            "start_column": "Actual_Start",
            "activity": "Change: Actual_Start"
          },
          {
            "start_column": "Actual_End",
            "activity": "Change: Actual_End"
          },
          {
            "start_column": "Planned_Start",
            "activity": "Change: Planned_Start"
          },
          {
            "start_column": "Planned_End",
            "activity": "Change: Planned_End"
          },
          {
            "start_column": "Requested_End_Date",
            "activity": "Change: Requested_End_Date"
          },
          {
            "start_column": "Change_record_Open_Time",
            "activity": "Change: record_Open_Time"
          },
          {
            "start_column": "Change_record_Close_Time",
            "activity": "Change: record_Close_Time"
          }
        ]
      }
    },
    {
      "label": "Incident_Activity",
      "id_column": "IncidentActivity_Number",
      "event": {
        "related_entities": ["Knowledge_Document", "
           Assignment_Group", "Incident", "Interaction"],
        "create_from": [
          {
            "start_column": "DateStamp",
            "end_column": "DateStamp",
            "activity": "Incident_Activity: {IncidentActivity_Type}
               "
          }
        ]
      }
    },
    {
      "label": "Change",
      "id_column": "ID",
      "event": {
        "related_entities": ["<Change_Activity:Service_Component",
           "<Change_Activity:Configuration_Item"],
        "create_from": [
          {
            "start_column": "Scheduled_Downtime_Start",
            "activity": "Change: Scheduled_Downtime_Start"
          },
          {
            "start_column": "Scheduled_Downtime_End",
            "activity": "Change: Scheduled_Downtime_End"
          }
        ]
      }
    },
    {
      "label": "Assignment_Group",
      "id_column": "ID"
    },
    {
      "label": "Knowledge_Document",
      "id_column": "ID"
    },
    {
      "label": "Configuration_Item",
      "id_column": "ID"
    },
    {
      "label": "Incident",
      "id_column": "Incident_ID",
      "event": {
        "related_entities": ["Service_Component", "
           Configuration_Item", "Knowledge_Document"],
        "create_from": [
          {
            "start_column": "Open_Time",
            "activity": "Incident: Open_Time"
          },
          {
            "start_column": "Reopen_Time",
            "activity": "Incident: Reopen_Time"
          },
          {
            "start_column": "Resolved_Time",
            "activity": "Incident: Resolved_Time"
          },
          {
            "start_column": "Close_Time",
            "activity": "Incident: Close_Time"
          }
        ]
      }
    },
    {
      "label": "Service_Component",
      "id_column": "ID"
    },
    {
      "label": "Interaction",
      "id_column": "Interaction_ID",
      "event": {
        "related_entities": ["Service_Component", "
           Configuration_Item", "Knowledge_Document"],
         "create_from": [
         {
           "start_column": "Open_Time_First_Touch",
           "activity": "Interaction: Open_Time_First_Touch"
         },
         {
           "start_column": "Close_Time",
           "activity": "Interaction: Close_Time"
         }
        ]
      }
    }
```

```
  ]
}
```

Listing B.1: BPI 14 transformation configuration

B.2 BPI 14 node degree Statistics

node type	**average**	**stdev**	**max**	**min**
Event	4.44	0.52	24	3
Entity	6.76	219.58	111222	0
Entity: Assignment_Group	1928.66	6193.02	84143	1
Entity: Change	11.45	41.60	5035	4
Entity: Change_Activity	6.78	0.62	7	4
Entity: Configuration_Item	41.65	448.15	22532	0
Entity: Incident	12.96	9.75	181	0
Entity: Incident_Activity	1	0	1	1
Entity: Interaction	5.12	7.02	180	2
Entity: Knowledge_Document	379.55	1097.3	23546	0
Entity: Service_Component	1928.61	7108.51	111222	0
LEvent	1.85	0.35	2	1
LEvent: Assignment_Group	1.99	0.02	2	1
LEvent: Change	1.91	0.28	2	1
LEvent: Change_Activity	1.85	0.35	2	1
LEvent: Configuration_Item	1.97	0.14	2	1
LEvent: Incident	1.9	0.26	2	1
LEvent: Incident_Activity	1	0	1	1
LEvent: Interaction	1.80	0.39	2	1
LEvent: Knowledge_Document	1.99	0.05	2	1
LEvent: Service_Component	1.99	0.022	2	1
Log	0	0	0	0

Table B.1: Full in-degree statistics for the BPI 14 event graph

node type	**average**	**stdev**	**max**	**min**
Event	0	0	0	0
Entity	0	0	0	0
Entity: Assignment_Group	0	0	0	0
Entity: Change	0	0	0	0
Entity: Change_Activity	0	0	0	0
Entity: Configuration_Item	0	0	0	0
Entity: Incident	0	0	0	0
Entity: Incident_Activity	0	0	0	0
Entity: Interaction	0	0	0	0
Entity: Knowledge_Document	0	0	0	0
Entity: Service_Component	0	0	0	0
LEvent	2.85	0.35	3	2
LEvent: Assignment_Group	2.997	0.022	3	2
LEvent: Change	2.91	0.28	3	2
LEvent: Change_Activity	2.85	0.35	3	2
LEvent: Configuration_Item	2.97	0.14	3	2
LEvent: Incident	2.926	0.26	3	2
LEvent: Incident_Activity	2	0	2	2
LEvent: Interaction	2.80	0.39	3	2
LEvent: Knowledge_Document	2.99	0.05	3	2
LEvent: Service_Component	2.99	0.02	3	2
Log	4926161	0	4926161	49261611

Table B.2: Full out- degree statistics for the BPI 14 event graph

Appendix C

BPI 17

C.1 BPI 17 configuration file

```
{
  "connection": {
    "neo4j": {
      "jdbc": "bolt://127.0.0.1:7687",
      "user": "neo4j",
      "password": "1234"
    }
  },
  "log": {
    "name": "BPI17"
  },
  "entity": [
  {
    "label": "application_events",
    "id_column": "ID",
    "event": {
      "related_entities": [">applications", ">resources", ">offers"
          ],
      "create_from": [
      {
        "start_column": "startTime",
        "end_column": "completeTime",

        "activity": "{Activity}"
      }
      ]
    }
  },
  {
    "label": "applications",
    "id_column": "ApplicationID"
  },
  {
    "label": "offer_events",
    "id_column": "ID",
    "event": {
      "related_entities": [">offers", ">resources", ">offers", ">
         offers:>applications"],
      "create_from": [
      {
        "start_column": "startTime",
        "end_column": "completeTime",

        "activity": "{Activity}"
      }
      ]
    }
  },
  {
    "label": "offers",
    "id_column": "OfferID"
  },
  {
    "label": "workflow_events",
    "id_column": "ID",
    "event": {
      "related_entities": [">workflows", ">resources", ">offers",
         ">offers:>applications"],
      "create_from": [
      {
        "start_column": "startTime",
        "end_column": "completeTime",

        "activity": "{Activity}"
      }
      ]
    }
  },
  {
    "label": "workflows",
    "id_column": "WorkflowID"
  },
  {
    "label": "resources",
    "id_column": "resource"
  }
  ]
}
```

Listing C.1: BPI 17 transformation configuration

C.2 BPI 17 node degree Statistics

node type	average	stdev	max	min
Event	3.85	0.54	5	3
Entity	2.35	128.87	138130	0
Entity: application_events	0.50	0.50	1	0
Entity: applications	34.44	15.90	176	8
Entity: offer_events	1.0	0.0	1	1
Entity: offers	20.60	13.86	134	2
Entity: resources	8002.79	12418.40	138130	14
Entity: workflow_events	1.0	0.0	1	1
Entity: workflows	5.54	3.63	118	1
LEvent	1.71	0.45	2	1
LEvent: application_events	1.0	0.0	1	1
LEvent: applications	1.97	0.16	2	1
LEvent: offer_events	1.0	0.0	1	1
LEvent: offers	1.95	0.21	2	1
LEvent: resources	1.99	0.01	2	1
LEvent: workflow_events	1.0	0.0	1	1
LEvent: workflows	1.81	0.38	2	1
Log	0.0	0.0	0	0

Table C.1: Full in-degree statistics for the BPI 17 event graph

node type	average	stdev	max	min
Event	0.0	0.0	0	0
Entity	0.0	0.0	0	0
Entity: application_events	0.0	0.0	0	0
Entity: applications	0.0	0.0	0	0
Entity: offer_events	0.0	0.0	0	0
Entity: offers	0.0	0.0	0	0
Entity: resources	0.0	0.0	0	0
Entity: workflow_events	0.0	0.0	0	0
Entity: workflows	0.0	0.0	0	0
LEvent	2.71	0.45	3	2
LEvent: application_events	2.0	0.0	2	2
LEvent: applications	2.97	0.16	3	2
LEvent: offer_events	2.0	0.0	2	2
LEvent: offers	2.95	0.21	3	2
LEvent: resources	2.99	0.01	3	2
LEvent: workflow_events	2.0	0.0	2	2
LEvent: workflows	2.81	0.38	3	2
Log	4466693.0	0.0	4466693	4466693

Table C.2: Full out-degree statistics for the BPI 17 event graph