MASTER

R2PG-DM
a direct mapping from relational databases to property graphs

Stoica, R.A.

Award date:
2019

Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Database Research Group

R2PG-DM: A direct mapping from relational databases to property graphs

Master Thesis

Radu-Alexandru Stoica (0929681)

Supervisors:
Dr. George H.L. Fletcher
Dr. Juan F. Sequda (data.world, USA)

Assessment Committee:
Dr. George H.L. Fletcher
Dr. Nikolay Yakovets
Dr. Dirk Fahland
Dr. Juan F. Sequda (data.world, USA)

Eindhoven, July 2019

R2PG-DM: A direct mapping from relational databases to property graphs

Author: Radu-Alexandru Stoica
Supervisors: George Fletcher, Juan F. Sequeda

Abstract— **The idea of mapping relational databases to graphs is not a new topic. However, many mappings are related to other types of graphs such as RDF. In this paper, we present R2PG-DM, which is a direct mapping from relational databases to property graphs. R2PG-DM focuses on the modelling of the property graph with respect to the relational schema. The semantics of this mapping is defined using Datalog. We analyze the mapping by defining two fundamental properties and two desirable properties. The two fundamental properties are information preservation and query preservation. We show that our mapping satisfies both fundamental properties. The desirable properties are monotonicity and semantics preservation. However, it is impossible to have a direct mapping satisfying all four properties due to monotonicity. [11]. Therefore, we show that our mapping is information preserving, query preserving and semantics preserving after suffering an extension. In the end, an implementation of R2PG-DM is presented as well.**

I. INTRODUCTION

Graph databases such as Neo4j are becoming very popular in industry nowadays, finding many useful use cases in many areas such as fraud detection, recommendation engines or privacy and risk compliance. Graph analytics are a very effective way of gaining insights from the graph-structured data collections such as social, financial or biological networks. A real-life case where graph analytics were used on property graphs is the Offshore Leaks financial social network data set, linking company officers and their companies registered in Bahamas.

However, most of the data found in the real world resides in relational databases. [11] Hence, making the graph analysis task difficult. By understanding how relational databases and property graphs relate to each other, a mapping can be defined that will allow the construction of property graphs using a relational database instance and schema as input.

In this paper, we define a direct mapping from relational databases to property graphs. A direct mapping automatically translates a relational database to property graphs, without any external interaction.

The study of direct mappings from relational to property graphs is not as well-developed. Of the small handful of approaches, the focus has been on optimized layout for efficient query processing or mappings which are lossy or obfuscate the input RDB schema. Furthermore, all current direct mappings are defined procedurally (i.e., defined with pseudo-code), making it difficult to formally reason about their correctness and other basic properties.

Our direct mapping is inspired by the approach of Sequeda et al. to direct mappings from RDB to RDF graphs. [11] We present a direct mapping which is: (1) domain and schema-independent, i.e., works regardless of the source database schema and instance, and (2) transforms the content of the source instance into a target instance., i.e., given a RDB instance generates a corresponding PG instance. A direct mapping is preferred as this type of solution is capable of translating data among different types of schema without any user interaction. Therefore, it is not required to understand the model of the final result before any interaction with the resulted model is even necessary. We study two properties which are fundamental to a direct mapping: information preservation and query preservation. Furthermore, we also study two desirable properties: monotonicity and semantics preservation.

Information preservation property assures that the original database can be reconstructed from the property graph resulted from the mapping. Query preservation assures that for every query over the relational database instance an equivalent query over the property graph obtained from the mapping exists and the output of the two queries is similar. Monotonicity implies that the updates to the database does not affect the generated property graph. Finally, a direct mapping is semantics preserving if the condition of a set of integrity constraints is reflected in the final result as well. Hence, if the input of the mapping is consistent, then the output of the mapping must be consistent as well and the same condition applies for the case when the input is inconsistent.

In this paper, we show that our mapping satisfies the fundamental properties and is also monotone in every scenario except when the database violates an integrity constraint, when our direct mapping generates a consistent property graph, which means it is not semantics preserving. Based on Sequeda et al. [11], we know that monotonicity is the obstacle in this case as no monotone direct mapping can be semantics preserving. Therefore, we extend out direct mapping in order to introduce an inconsistency in the generated property graph when an integrity constraint is violated in the relational database. Finally, we show that the extended mapping is information preserving, query preserving, non-monotone and semantics preserving.

Finally, the paper presents an implementation of the direct mapping R2PG-DM. Also, several use cases where relational databases such as Github or Reddit are translated into property graphs are shown.

Also, a summary of this paper exists. The paper has been presented at AMW 2019 conference in Paraguay on June 6^{th}

2019. [12]

II. Related work

In this section, we investigate several papers on the topic of direct mappings from relational databases to property graphs. We give an insight of each mapping, we analyze how each mapping works and we try to identify the issues.

First, we show a procedural mapping called RDB2Graph. [14] Given a relational database D and a set of modeling rules Γ, the mapping generates a directed graph (V, E), where V is the set of vertices and E is the set of edges. The mapping consists of two steps. Phase 1 discovers relationships within each tuple, while the second phase establishes additional relationships among the vertices constructed from the first phases. In the first phase, pairs of attributes (C_s, C_t) are used from a relation r, which are mapped to a set of keys (K_s, K_t) using the corresponding value u. For each c in C_s or C_t, the corresponding value u in each tuple is paired with the corresponding k in K_s or K_t. This forms a key-value pair (K, u), which is added to the generated source or target vertex. A bit (b) is also used to indicate whether the edge is bidirectional or not. A selection predicate (p) is used to filter tuples. At the end of phase 1, a set (V, E), representing the graph, is generated and passed to phase 2. In phase 2, the framework generates additional edges based on two sets of vertex schema (K_s, K_t), the bit (b) to specify if the generated edge is bidirectional or not and a vertex selection predicate (q) to filter vertices. Given the vertices, the framework further filters them using vertex selection predicate q. The second phase produces a new graph (V, E) constructed with the additionally generated edges.

Second, we investigate the mapping R2G created by De Virgilio et al. [13]. This method uses four main components which construct the mapping. This is a mapping from relation database to property graphs which aggregates data that is most likely to occur together in a single node. First, a schema is created based on several rules. Therefore, a schema graph RG for a relational schema R is a directed graph $\langle N, E\rangle$ such that every attribute of a relation R is a node in N and there is an edge $(A_i, A_j) \in E$ if one of the following holds: (a) A_i belongs to a key of a relation r in R and A_j is a non-key attribute of r, (b) A_i, A_j belong to a key of a relation r in R, (c) A_i, A_j belong to R_i and R_j respectively and there is a foreign key between A_i and A_j. The second component is called Data mapping (DM). A property graph is created at this step. The main idea is aggregating values of different tuples in the same node. Data values that are likely to be retrieved together in the evaluation of queries are stored in the same node. In order to avoid the overload of a node, the $unifiable$ property is used. This is defined as follows: Two data values $v_1 = t_1[A]$ and $v_2 = t_2[B]$ are $unifiable$ in a relational database r if one of the following holds: (i) $t_1 = t_2$ and both A and B do not belong to a multi-key; (ii) t_1 and t_2 are join-able and A belongs to a multi-key; (iii) t_1 and t_2 are join-able, A and B do not belong to a multi-key and there is no other tuple t_3 that is join-able with t_2. The other two components are a Query Mapper (QM), which translates conjunctive SQL queries into a XQuery-like query language. The fourth component is called Graph Manager (GM), which migrates the data and executes queries over the target database by using the mappings generated by DM and the QM.

Finally, we investigate a full-property mapping where the method takes advantage of edge properties as well. This is a fully procedural mapping presented by Orel et al. [9]. The algorithm is composed of five parts as follows: (1) the migration order is defined. The tables which reference two other tables and are not referenced by any other table will be converted into edges at the end. For each tuple, the tables that do not reference any other tables are converted into nodes. Next, tables which reference other already converted tables are converted into nodes as well. At this point the foreign keys are also converted into edges. Because, all the tables with references to other tables have to already be converted, the algorithms also need to check for cyclic references among tables. Therefore, the second (2) and third (3) components are represented by algorithms to find cyclic references for a table and finding all cyclic references in the whole schema. The next step (4) is an algorithm for converting the data from a relational table to graph elements representing tuples being added to the graph database. The nodes are labeled as the table's name, and it gets an id which is the concatenation of all the attributes composing the primary key. The non-key attributes are converted into node properties. The newly created edges are labeled with the table name and the given id which corresponds to the primary key value. All non-key attributes in a tuple are added as edge properties. The final and main algorithm (5), is the algorithm that iterates over all the relational tables in the relational schema and performs the steps (1),(2),(3) and (4). The output of the main algorithm (5) where all the previous algorithms are brought together is a graph database instance G.

After investigating all the previous mappings, it can be observed that the authors have developed the mappings with the idea of query efficiency in mind. All these papers try to optimize the query execution over the generated graphs. However. the trade-off in this case is the modelling which has to suffer. The generated graphs are either too complex, lossy or obfuscate the input RDB schema. By being procedural and by aggregating data, one is not able to reason over properties such as information preservation, query preservation or semantics preservation. Therefore, proving the correctness of these mappings is difficult.

Therefore, in this paper we follow a different approach where we give a direct mapping which follows a natural, logical translation of relation databases to property graphs by preserving the schema and by reasoning over basic properties of a direct mapping.

III. Preliminaries

In this section, we define the basic terminology used in the paper.

A. Relational databases

Let $\mathcal{D}$ be a countably infinite domain, $NULL$ a reserved symbol with the condition that $NULL \notin \mathcal{D}$, $\mathcal{A}$ a countably infinite domain of attributes names and rId a reserved symbol with the condition that $rId \notin \mathcal{A}$. A $schema$ $\mathbf{R}$ is a finite set of relation names, where for each $R \in \mathbf{R}$, $att(R)$ denotes the nonempty finite set of attributes names associated to R. Each $att(R)$ contains the attribute rId. For each attribute a from set $att(R) \setminus \{rId\}$, there is a condition that $A \in \mathcal{A}$. An instance I of $\mathbf{R}$ assigns to each relation symbol $R \in \mathbf{R}$ a finite set $R^I = \{t_1, ..., t_l\}$ of tuples, where each tuple $t_j (1 \leq j \leq l)$ is a function that assigns to each attribute in $att(R) \setminus \{rId\}$ a value from $\mathcal{D} \cup \{NULL\}$. $t.A$ refer to the value of a tuple t in an attribute A. We call the value $t.rId$ the identifier of tuple t such that for all i, j it satisfies $t_i.rId \neq t_j.rId$, where $i \neq j$. Also, for all i, it satisfies $t_i.rId \in \mathbb{N}$.

Relational algebra: In this paper, we consider relational database containing null values. Hence, we use a modified version of relational algebra as a query language as defined by Sequeda et al. [11]. Therefore, we refer the reader to this paper for more details.

B. Property graphs

Let $\mathcal{P}$ be an infinite set of property names, $\mathcal{V}$ an infinite set of atomic values, $\mathcal{L}$ an infinite set of labels (for nodes and edges) and $\mathcal{T}$ an infinite set of datatypes. We give a definition of a property graph as a mixture of two definitions given by Angles et al. and Bonifati et al. [2] [3]

***Definition** 1:* A property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$, where:

1) N is a finite set of nodes.
2) E is a finite set of edges. E has no common elements with N.
3) $\rho : E \rightarrow (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N.
4) $\lambda : (N \cup E) \rightarrow P(L)$ is a possible function that associates a node/edge with a set of labels from L. ($P(S)$ is a set of finite subsets of S)
5) $\sigma : (N \cup E) \times P \rightarrow V$ is a partial function that associates nodes/edges with properties. N and E are disjoint.

C. G-CORE

In this paper, we use G-CORE [2] as a query language for property graphs. G-CORE is a a mature query language which support several advanced features such as multi-graph queries, joins, sub-queries or views. G-CORE enhances graph analysis over property graphs by introducing paths as first-class citizens alongside nodes and edges. G-CORE is built for Path Property Graphs (PPG). However, the G-CORE applies very well to the property graph defined in this paper, as a PPG is only an extension of a property graph where paths are treated as first-class citizens alongside nodes and edges. [2] The syntax of G-CORE considers operators such as *CONSTRUCT*, *MATCH*, *OPTION*, *WHERE* for constructing graphs. It also supports set operations such as *UNION*, *INTERSECT*, *MINUS*.

For the purpose of this paper, we are not only interested in constructing the property graph, but we also want to retrieve the property graph data in order to compare the query results for relational databases with the query results for property graphs. Hence, we use an extension of G-CORE which introduces the operator *SELECT*. The *SELECT* operator projects tabular results for the property graphs.

As described by Angles et al., the semantics of G-CORE will be denoted by $[\![.]\!]_{\Omega,\mathcal{G}}$ throughout the paper, where Ω is a set of bindings which makes reference to objects from the property graph $\mathcal{G}$.

Here we only present a brief summary of the potential of G-CORE query language. However, G-CORE is much more extensive. Therefore, for more details regarding the syntax and semantics of G-CORE, we refer the reader to the paper of Angles et al. [2]

IV. Direct mapping: Definition and properties

A direct mapping translates relational databases into property graphs without any input from the user. The input of a direct mapping $\mathcal{M}$ is a relational schema $\mathbf{R}$, its corresponding instance I and a set Σ of PKs and FKs over $\mathbf{R}$. The output is a property graph. Furthermore, let $\overline{FK}$ be the size of foreign keys in Σ and $\overline{R}$ be the size of relation names in $\mathbf{R}$.

Let $\mathcal{G}$ be the set of all property graphs and $\mathcal{RC}$ the set of all triples of the form $(\mathbf{R}, \Sigma, I)$ where $\mathbf{R}$ is a relational schema, I is its corresponding instance and Σ is a set of PKs and FKs over $\mathbf{R}$.

***Definition** 2:* (Direct mapping) A direct mapping $\mathcal{M}$ is a total function from $\mathcal{RC}$ to $\mathcal{G}$.

Next, we introduce the properties of the direct mapping. There are two fundamental properties: information preservation and query preservation; and two desirable properties: monotonicity and semantics preservation. The fundamental properties are important because they assure that the data is unaltered after the direct mapping is applied. For this reason, we decided to create a schema of the relational database in order to guarantee this property. From our knowledge, there is no standard schema language for property graphs. Therefore, we encoded the schema as a property graph as well. Also, the query preservation guarantees that every relational algebra query over the relational database has a G-CORE query equivalent over the resulted property graph. This fundamental property is very powerful as it assures that G-CORE queries can be translated over other more popular query languages used in real-life such as SQL. In addition, we have monotonicity as a desirable property. This would ensure that any update to the relational database will have a minimal impact on the mapping. Finally, the semantics preservation property ensures that any violated integrity constraints of relational databases are also reflected as inconsistency over the resulted property graph.

A. *Fundamental properties*

1) Information preservation: A direct mapping is information preserving if no information about the relational instance being translated is lost during the mapping process. Let $\mathcal{I}$ be the set of all possible relational instances. Then we have that:

***Definition** 3:* (Information preserving) A direct mapping $\mathcal{M}$ is information preserving if there is a computable mapping $\mathcal{N} : \mathcal{G} \rightarrow \mathcal{I}$ such that for every relational schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$ and instance *I* of $\mathbf{R}$ satisfying $\Sigma : \mathcal{N}(\mathcal{M}(\mathbf{R}, \Sigma, I)) = I$.

2) Query preservation: A direct mapping is query preserving if every query over a relational database can be translated into an equivalent query over the resulted property graph.

In order to define query preservation, we use relational algebra in order to express relational queries and we use G-CORE to express property graphs queries. We have already introduced in section III-A a modified relational algebra that can express null values as well and an extended G-CORE query language for property graphs in section III-C. However, the problem is that there is a difference between the output generated by relational algebra and G-CORE. Therefore, the tuples returned by relational algebra must be converted to match the mappings returned by G-CORE. For this case, we use a function tr. Therefore, let $\mathbf{R}$ be a relation schema, R be a relation name in $\mathbf{R}$, I an instance of $\mathbf{R}$ and t a tuple in R^I. The function $tr(t)$ is defined as the mapping μ such that: (1) the domain of μ is $\{A | A \in att(R)$ and $t.A \neq NULL\}$. and (2) $\mu(A) = t.A$ for every A in the domain of μ.

***Example** 1:* Assume that a relational schema contains a relation name $PERSON$ and attributes $NAME$, and DoB. Moreover, assume that t is a tuple in this relation such that $t.NAME =$Sue, $t.DoB =$NULL. Then, $tr(t) = \mu$, where the domain of μ is $\{NAME\}$, $\mu(NAME) =$ Sue.

***Definition** 4:* A direct mapping $\mathcal{M}$ is query preserving if for every relational schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$ and relational algebra query Q over $\mathbf{R}$, there exists a G-CORE query Q^* with a set of bindings Ω corresponding to the content of the $MATCH$ clause over the graph $\mathcal{M}(\mathbf{R}, \Sigma, I)$ such that for every instance I of $\mathbf{R}$ satisfying $\Sigma : tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega, \mathcal{M}(\mathbf{R}, \Sigma, I)}$.

B. *Desirable properties*

1) Monotonicity: Let I_1 and I_2 be two database instances over a relational schema $\mathbf{R}$. I_1 is a subset of I_2 ($I_1 \subseteq I_2$), if for every relation name $R \in \mathbf{R}$, the set of tuples of R for instance I_1 is a subset of the set of tuples of R for instance I_2. We call a direct mapping $\mathcal{M}$ monotone if for any pair of instances (I_1, I_2), the property graph resulted from the instance I_1 is a subset of the property graph resulted from the instance I_2. Hence, any alteration performed on the database must not affect any of the computed elements during the mapping.

***Definition** 5:* A direct mapping $\mathcal{M}$ is monotone if for every relational schema $\mathbf{R}$, set Σ of PK's and FK's over $\mathbf{R}$ and instances I_1, I_2 of $\mathbf{R}$ such that $I_1 \subseteq I_2 : \mathcal{M}(\mathbf{R}, \Sigma, I_1) \subseteq \mathcal{M}(\mathbf{R}, \Sigma, I_2)$.

2) Semantics preservation: A direct mapping is considered semantics preserving if the mapping reflects the condition of a set of PKs and FKs defined over a relational schema. Therefore, given a relational schema $\mathbf{R}$, a set Σ of PKs and FKs over $\mathbf{R}$ and its corresponding instance I, a mapping is semantics preserving if it generates a consistent property graph from I when the set Σ is consistent, and it generates an inconsistent property graph otherwise.

Let r be a relation name in I, pk the primary key of r and fk a foreign key from r to a relation name s, where $s \in I$. Also, let a be an attribute in r and b an attribute in s, where a and b are part of the foreign key fk. A set of PKs and FKs Σ is considered inconsistent when at least one of the following rules is not satisfied for an instance I: (1) pk contains a NULL value, (2) pk contains repetitive values (duplicates), (3) a value v_a in a, $(a \in r)$ does not exist in the attribute b, $(b \in s)$.

***Definition** 6:* A direct mapping $\mathcal{M}$ is semantics preserving if for every relation schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$ and instance I of $\mathbf{R}$: $I \models \Sigma$ iff $\mathcal{M}(\mathbf{R}, \Sigma, I)$ is consistent.

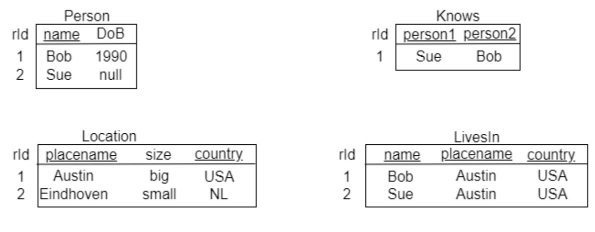
V. THE DIRECT MAPPING DM

We introduce a direct mapping $\mathcal{DM}$ inspired from the direct mapping from RDB to RDF proposed by Sequeda et al. [11]. $\mathcal{DM}$ is defined as a set of Datalog [1] rules, which are divided in two parts: translate relational instances and translate relational schemas. For more information regarding

In Section A, we present the relational instance used as example in this paper. In Section B, we present a function, called id, which will generate a global unique identifier for all nodes and edges resulted from the mapping. In Section C, we present the predicates used to store the relational database, the input of $\mathcal{DM}$. In Section D, we present the predicates that compose the actual mapping which generates the property graph based on the relational database. Finally, in Section E, we show how the relational schema is translated as well, encoded as a property graph using Datalog rules.

A. *Running instance*

Throughout this paper, we use the following instance as a running example.



Person

rId	name	DoB
1	Bob	1990
2	Sue	null

Knows

rId	person1	person2
1	Sue	Bob

Location

rId	placename	size	country
1	Austin	big	USA
2	Eindhoven	small	NL

LivesIn

rId	name	placename	country
1	Bob	Austin	USA
2	Sue	Austin	USA

Fig. 1. Instance

Notice the insert in the schema of the hidden field rId for each tuple of every table. In any real-life relational

[1]We refer the reader to [1] for the syntax and semantics of Datalog.

database system, for each row insertion, a row identifier is automatically assigned by the RDBMS system. An example can be seen in the PostgreSQL relational database, where the row identifier is specified in the documentation. [4]

We have the following constraints about the schema: *name* is the primary key of *Person*, *(person1,person2)* is the composed primary key of *Knows*, *(placename,country)* is the composed primary key of *Location*, *(name,placename,country)* is the composed primary key of *LivesIn*, *person1* is a foreign key in *Knows* that references attribute *name* in *Person*, *person2* is a foreign key in *Knows* that references attribute *name* in *Person*, *name* is a foreign key in *LivesIn* that references attribute *name* in *Person* and *(placename,country)* is a composed foreign key in *LivesIn* that references attributes *(placename,country)* in *Location*.

B. The id function

Each property graph element (node or edge) requires an unique identifier in order to construct a sound property graph. Therefore, an injective function $id : (\mathbb{N} \cup \{NULL\}) \times (\mathbf{R} \cup \{NULL\}) \times (\mathcal{A} \cup \{NULL\}) \times (\mathbb{N} \cup \{NULL\}) \times (\mathbf{R} \cup \{NULL\}) \times \bigcup_{i \geq 0} (\mathcal{A} \cup \{NULL\} \times \mathcal{A} \cup \{NULL\}) \rightarrow \mathbb{N}$ is assumed to be given for generating unique identifiers using tuple identifiers, relation symbols and attributes from the schema $\mathbf{R}$. The id function can have many variations as we will see in Section D and Section E in this Chapter. The translation of relational instances section uses two variations: (1) $(\mathbb{N} \times \mathbf{R}) \rightarrow \mathbb{N}$ for nodes, and (2) $(\mathbb{N} \times \mathbf{R} \times \mathbb{N} \times \mathbf{R} \times \bigcup_{i \geq 0} (\mathcal{A} \times \mathcal{A})) \rightarrow \mathbb{N}$ for edges. The translation of relational schemas uses five variations: (1) $\mathbf{R} \rightarrow \mathbb{N}$, (2) $(\mathbf{R} \times \mathcal{A}) \rightarrow \mathbb{N}$, (3) $(\mathbf{R} \times \mathbf{R} \times \mathcal{A} \times \mathcal{A}) \rightarrow \mathbb{N}$, (4) $(\mathbf{R} \times \mathbf{R} \times \bigcup_{i \geq 0} (\mathcal{A} \times \mathcal{A})) \rightarrow \mathbb{N}$, and (5) $(\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$.

Several examples for using the id function are presented below:

***Example** 2:* We show three different examples:

$id(2, Person) = 2$, It generates the unique global identifier with value 2 for the node generated from the second tuple in relation *Person*.

$id(1, Location) = 4$, It generates the unique global identifier with value 4 for the node generated from the first tuple in relation *Location*.

$id(2, Person, 2, LivesIn, \{name\}, \{name\}) = 11$, It generates the unique global identifier with value 11 for the edge generated from the second tuple in relation *Person* to the second tuple in relation *LivesIn* induced by the FK *name* in relation *LivesIn* to the primary key *name* in relation *Person*.

C. Storing relational databases

The direct mapping $\mathcal{DM}$ is declaratively specified as a set of Datalog rules. Therefore, the input $(\mathbf{R}, \Sigma, I)$ has to be encoded to Datalog as well. Therefore, the following predicates are used to store a relation schema $\mathbf{R}$, a set of PKs and FKs Σ over $\mathbf{R}$ and the values of each tuple t, where $t \in r$ and $r \in \mathbf{R}$:

- $REL(r)$: Indicates that r is a relation name in $\mathbf{R}$.
- $ATTR(a, r)$: Indicates that a is an attribute in the relation r in $\mathbf{R}$.
- $PK_n(a_1, ..., a_n, r)$: Indicates that $r[a_1, ...a_n]$ is a primary key in Σ.
- $FK_n(a_1, ...a_n, r, b_1, ...b_n, s)$: Indicates that $r[a_1, ...a_n] \subseteq s[b_1, ...b_n]$ is a foreign key in Σ.
- $VALUE(v, a, t.rId, r)$: Indicates that v is the value of an attribute a in a tuple with identifier $t.rId$ in a relation r from $\mathbf{R}$.

We also use the definition of $CONCAT_n(x_1, ..., x_n, y)$ which holds if y is the concatenation of the strings $x_1, ..., x_n$. We refer the reader to the Appendix I.A, for a concrete example of this encoding.

D. Translating a relational database into property graph

We now introduce the rules that translate a relational database into a property graph.

1) Generating nodes: First, we show how nodes are generated. For this, we define a Datalog rule which converts every tuple of every relation name r in $\mathbf{R}$ into a node in the property graph. Each node has a label represented by the relation name r and a unique identifier generated by the id function. Rule 1 from Table I presents this rule.

2) Generating edges: Second, we generate edges based on foreign keys among the relation names in $\mathbf{R}$. Each edge contains an unique identifier generated by the id function and a label represented by the concatenation between the relation name r, the symbol $'-'$, and relation name s. Therefore, for each foreign key from r to s in $\mathbf{R}$ we apply rule number 2 from the Table I.

3) Generating properties: Finally, we generate properties for nodes based on the attributes of a relation name r in $\mathbf{R}$. Each property is assigned to a node using the unique identifier and has the format: "$\{0\}$: $\{1\}$", where $\{0\}$ is the attribute name and, $\{1\}$ is the attribute value. Therefore, for each attribute a of a relation name r in $\mathbf{R}$ we apply rule 3 from Table I.

Here we have presented the Datalog rules used to generate a property graph from a relational database. For a concrete example of how the rules are applied on the presented running instance, we refer the reader to the Appendix I.B.

E. Translating a relational schema into property graph

In this section, we represent several Datalog rules for encoding a relation schema using Datalog rules. As there is no schema language standard for property graphs at the moment, we decided that encoding the schema as a property graph is the best option as it serves the purpose of this paper which is proving the information preserving property in section VI-A. For simplicity, we decided to encode the schema as a property graph using Datalog rules as well. Hence, no new schema language has to be defined.

Our schema has four types of nodes which can have properties and be connected by edges. There are four types of nodes with respect to their label in our schema as follows: (1) Rel nodes representing a relation name r from $\mathbf{R}$, (2) Att nodes representing an attribute a of a relation name r

TABLE I

DATALOG RULES FOR GENERATING PROPERTY GRAPHS

Rule #	Head	Body
1	$NODE(id(t.rId, r), r)$	$REL(r), VALUE(v_1, a_1, t.rId, r), ..., VALUE(v_n, a_n, t.rId, r)$
2	$EDGE_n(id_n(t_r.rId, r, t_s.rId, s, \bar{a}, \bar{b}), id(t_r.rId, r), id(t_s.rId, s), c)$	$FK_n(a_1, ..., a_n, r, b_1, ..., b_n, s), VALUE(v_1, a_1, t_r.rId, r), ..., VALUE(v_n, a_n, t_r.rId, r), VALUE(v_1, b_1, t_s.rId, s), ..., VALUE(v_n, b_n, t_s.rId, s), CONCAT_3(r, "-", s, c)$
3	$PROPERTY(id(t.rId, r), a, v)$	$NODE(id(t.rId, r), r), VALUE(v, a, t.rId, r)$

from **R**, (3) fk nodes representing a foreign key between two attributes a_r of relation name r and a_s of relation name s, where $r, s \in$ **R**, and (4) Fk nodes representing a foreign key among attributes $a_{r_1}, ..., a_{r_n}$ of relation name r and $a_{s_1}, ..., a_{s_n}$ of relation name s, where $r, s \in$ **R**.

There are three types of edges with respect to their label names: (1) $Rel - Att$ edges representing an edge between two nodes with labels Rel and Att, (2) $Att - fk$ edges representing an edge between two nodes with labels Att and fk, and (3) $fk - Fk$ edges representing an edge between two nodes with label fk and Fk.

The schema contains properties for nodes of type Rel and Att. There is only one property for nodes of type Rel: (1) a property $name$ storing the name of a relation name r from **R**. There are two properties for nodes of type Att: (1) a property $name$ which stores the attribute name of an attribute a of a relation name r from **R** and (2) a property pk, which stores a boolean value specifying if the current attribute name is part of the primary key of a relation name r from **R**. Next, we present the Datalog rules which generate all the schema elements.

1) Generating nodes: The following rules are used for generating nodes. First, we have the rule number 4 from Table II for generating nodes of type Rel. The node gets an id based on the relation name r in **R** and the label "Rel". Next, we have the rule number 5 from Table II for generating nodes of type Att, where we use the relation name r in **R** and an attribute a in r for generating the id. Next, we have the rule for generating nodes of type fk. For each foreign key $fKey$, we generate a node fk for each attribute composing $fKey$ from a relation name r to a relation name s, where $r, s \in$ **R**. This rule is present in Table II as rule number 6. Finally, we have the rule for generating nodes of type Fk. We generate a node for each foreign key from a relation name r to a relation name s, where $r, s \in$ **R**. The rule can be seen in Table II as rule 7.

2) Generating edges: Next, we generate the edges among all nodes in the schema. First, we generate edges of type $Rel - Att$ among all nodes of type Rel and nodes of type Att described in Table II as rule number 8. Next, rule 9 in Table II generates edges of type $Att - fk$ among all nodes of type Att and nodes of type fk for all attributes composing a foreign key between relations r and s. Finally, we generate edges of type $fk - Fk$ among all nodes of type fk and nodes of type Fk depicted by rule number 10 in Table II for each foreign key between relations r and s.

3) Generating properties: Furthermore, we generate properties for nodes of type Rel and Att. First, rule number 11 from Table II is used for generating the property $name$ for nodes of type Rel, where we store the relation name in the node of type Rel. Finally, we generate properties for nodes of type Att. A property $name$ is generated by rule 12 from Table II, which stores the attribute name and rule number 13 from Table II generates a property pk which specifies if the current attribute is part of the primary key of a relation name r from **R**.

Figure 2 presents the extracted schema for the running example. The nodes are colored differently for the sake of readability. Nodes of type Rel are colored blue, nodes of type Att are colored green, nodes of type fk are colored orange, and nodes of type Fk are colored red:

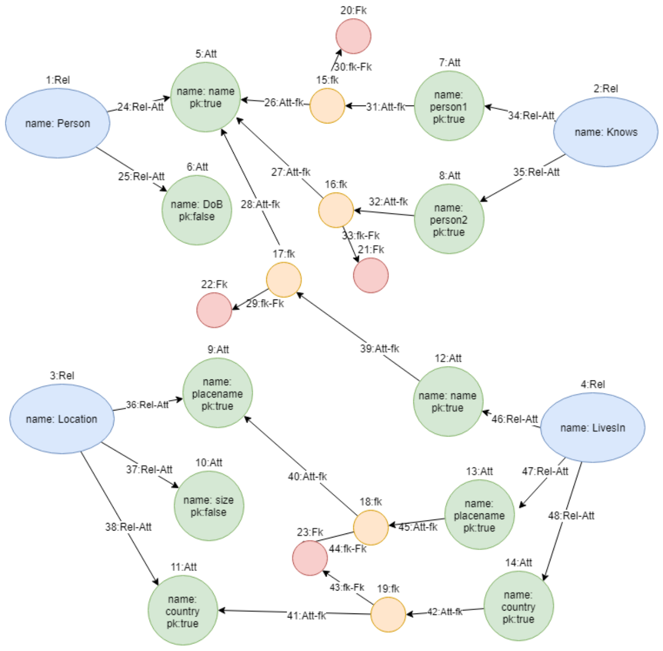


Fig. 2. Translation of running example

VI. PROPERTIES OF DM

We now study the direct mapping $\mathcal{M}$ with respect to the two fundamental properties: information preservation and query preservation and two desirable properties: monotonicity and semantics preservation defined in Chapter 4.

A. *Information preservation of $\mathcal{DM}$*

First, we reason about the information preservation of $\mathcal{DM}$. That is, we show that no information is lost during mapping:

TABLE II

DATALOG RULES FOR ENCODING A RELATIONAL SCHEMA TO PROPERTY GRAPH

Rule #	Head	Body
4	$NODE(id(r), "Rel")$	$REL(r)$
5	$NODE(id(r,a), "Att")$	$REL(r), ATTR(a,r)$
6	$\bigcup_{n=0}^{\overline{FK}} (\bigcup_{i=1}^{n} NODE_n(id(r,s,a_i,b_i), "fk")$	$FK_n(a_1,...,a_i,...,a_n,r,b_1,...,b_i,...b_n,s))$
7	$\bigcup_{n=0}^{\overline{FK}} NODE_n(id(r,s,a_1,...,a_n,b_1,...,b_n), "Fk")$	$FK_n(a_1,...,a_n,r,b_1,...,b_n,s)$
8	$EDGE(id(id(r),id(r,a)),id(r),id(r,a), "Rel-Att")$	$REL(r), ATTR(a,r)$
9	$\bigcup_{n=0}^{\overline{FK}} (\bigcup_{i=1}^{n} EDGE_n(id(id(r,a_i),id(r,s,a_i,b_i)),id(r,a_i),$ $id(r,s,a_i,b_i), "Att-fk")$	$FK_n(a_1,...,a_i,...,a_n,r,b_1,...,b_i,...b_n,s))$
10	$\bigcup_{n=0}^{\overline{FK}} (\bigcup_{i=1}^{n} EDGE_n(id(id(r,s,a_i,b_i),id(r,s,a_1,...,a_n,b_1,...,b_n))$ $,id(r,s,a_i,b_i),id(r,s,a_1,...,a_n,b_1,...,b_n), "fk-Fk")$	$FK_n(a_1,...,a_n,r,b_1,...,b_n,s))$
11	$PROPERTY(id(r), "name", r)$	$REL(r)$
12	$PROPERTY(id(r,a), "name", a)$	$REL(r), ATTR(a,r)$
13	$\bigcup_{n=0}^{\overline{R}} (\bigcup_{i=1}^{n} PROPERTY_n(id(r_n,a_i), "pk", "true")$	$PK_n(a_1,...,a_i,...,a_n,r_n))$

***Theorem** 1:* The direct mapping $\mathcal{DM}$ is information preserving.

The proof of this theorem involves providing a computable mapping $\mathcal{N}: \mathcal{G} \to \mathcal{I}$ that satisfies the condition in Definition 3. $\mathcal{N}$ has to be a computable mapping that can reconstruct the initial relational instance from the generated property graph. Therefore, for this reason we use the schema presented in section V-E. Therefore, by using the generated schema, we are able to reconstruct the relational instance schema. Finally, the tuples are obtained using the rules for generating the mapping described in section V-D. For more details of the proof, we reference the reader to the Appendix II.

B. Query preservation of $\mathcal{DM}$

Second, we show that every relational algebra query over the relational database has an equivalent G-CORE query over the generated property graph through the mapping $\mathcal{DM}$.

***Theorem** 2:* The direct mapping $\mathcal{DM}$ is query preserving.

The proof is complex as we have to consider every operator supported by the relational algebra query language. Therefore, the complete proof can be found in the Appendix. However, we are giving an outline of the proof in this section. The goal of the proof is to establish the relation mentioned in Definition 4. Assume given a relational schema **R** and a set Σ of PKs and FKs over **R**. Then we have to show that for every relational algebra query Q over **R**, there exists a G-CORE query Q^* such that for every instance I of **R** (possibly including null values) satisfying Σ:

$$tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega, \mathcal{M}(\mathbf{R}, \Sigma, I)} \ (1)$$

Here we present a recursive algorithm for translating a relation algebra query Q into an equivalent G-CORE query Q^*. Therefore, we prove that condition (1) holds using induction on the structure of Q. In order to show how the algorithm works, we will use the presented running example from section V-A and the relational algebra query $\sigma_{name=Bob}(Person) \bowtie LivesIn$.

The inductive proof starts by considering the two base relational algebra queries: the identity query R, where R is a relation name in **R**, and the query $NULL_A$, representing the empty relation. Therefore, the following two base cases are identified. For the sake of readability we denote the mapping $\mathcal{DM}(\mathbf{R}, \Sigma, I) = \mathcal{G}$ throughout the proof:

Non-empty relations: Assume that Q is the identity relational algebra query R, where $R \in$ **R** is a non-empty relation. Moreover, assume that $att(R) = \{A_1, ..., A_l\}$. Then a G-CORE query Q^* satisfying (1) is constructed as follows:

SELECT $n.A_1, ..., n.A_l$
MATCH$(n : R)$
ON $\mathcal{G}$

Notice that NULL values are also allowed as no extra checking is made in the query for requested attributes $n.A_1, ..., n.A_l$. In our example, the relation name *Person* is a non-empty relation. The equivalent G-CORE query is generated with input *Person*:

SELECT $n.name, n.DoB$
MATCH$(n : Person)$
ON $\mathcal{G}$

Empty relation: Assume that $Q = NULL_A$, and define $Q^* = SELECT{*}\ MATCH()\ ON\ \mathcal{G}$. It is trivially to show that condition (1) holds as by the definition of the function tr, the $NULL$ values are not being translated.

We now present the inductive step in the proof of Theorem 2. Assume that the theorem holds for relational algebra queries Q_1 and Q_2. That is, there exists G-CORE queries Q_1^* and Q_2^* such that:

$$tr(\llbracket Q_1 \rrbracket_I) = \llbracket Q_1^* \rrbracket_{\Omega, \mathcal{G}} (2)$$
$$tr(\llbracket Q_2 \rrbracket_I) = \llbracket Q_2^* \rrbracket_{\Omega, \mathcal{G}} (3)$$

We present equivalent relational algebra on property graphs queries for the following relational algebra operators: selection (σ), projection (π), rename (δ), join ($\bowtie$), union ($\cup$), and difference ($\setminus$).

Selection: We need to consider four cases to define query Q^* satisfying condition (1). In all these cases, we use the induction hypothesis (2). Also, we assume that $r \in \mathbf{R}$ and also $r \in Q_1$ in the following cases:

1) If Q is $\sigma_{A_1=a}(Q_1)$, then

Q^* = SELECT $n.A_1$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
WHERE $n.A_1 = a$
  AND $a \neq NULL$

2) If Q is $\sigma_{A_1 \neq a}(Q_1)$, then

Q^* = SELECT $n.A_1$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
WHERE $n.A_1 \neq NULL$
  AND $n.A_1 \neq a$

3) If Q is $\sigma_{isNull(A_1)}(Q_1)$, then

Q^* = SELECT $n.A_1$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
WHERE $n.A_1 = NULL$

4) If Q is $\sigma_{isNotNull(A_1)}(Q_1)$, then

Q^* = SELECT $n.A_1$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
WHERE $n.A_1 \neq NULL$

Following our example, we have that the following G-CORE query is generated with input $\sigma_{name=Bob}(Person)$:

SELECT $n.name$
MATCH($n : Person$)
ON $\mathcal{G}$
WHERE $n.name = Bob$

Projection: Assume that $Q = \pi_{\{A_1,...,A_l\}}(Q_1)$. Also, we assume that $r \in \mathbf{R}$ and also $r \in Q_1$ in the following cases. Then query Q^* satisfying condition (1) is defined as:

Q^* = SELECT $n.A_1, ..., n.A_l$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)

Rename: Assume that $Q = \delta_{A_1 \to B_1}(Q_1)$ and $att(Q) = \{A_1, ..., A_l\}$. Also, we assume that $r \in \mathbf{R}$ and also $r \in Q_1$ in the following cases. Then query Q^* satisfying condition (1) is defined as:

Q^* = SELECT $n.A_1$ AS $B_1, ..., n.A_l$
MATCH($n : r$)
ON $\mathcal{G}$
OPTIONAL (Q_1^*)

Join: Assume that $Q = (Q_1 \bowtie Q_2)$. Then query Q^* satisfying condition (1) is defined as:

Q^* = (Q_1^* INTERSECT Q_2^*)

Union: Assume that $Q = (Q_1 \cup Q_2)$. Then query Q^* satisfying condition (1) is defined as:

Q^* = (Q_1^* UNION Q_2^*)

Difference: We conclude our proof by assuming that $Q = (Q_1 \setminus Q_2)$. In this case we are using induction hypothesis (2) and (3). Then query Q^* satisfying condition (1) is defined as:

Q^* = (Q_1^* MINUS Q_2^*)

The complete proof can be found in the Appendix II. Now, we are going to show the complete translation of the relational algebra query mentioned in the beginning of this section $\sigma_{name=Bob}(Person) \bowtie LivesIn$:

SELECT $n_1.name$
MATCH ($n_1 : Person$)
ON $\mathcal{G}$
WHERE $n_1.name = Bob$
INTERSECT
SELECT $n_2.name, n_2.placename,$
    $n_2.country$
MATCH ($n_2 : LivesIn$)
ON $\mathcal{G}$
WHERE $n_2.name \neq NULL$

C. *Monotonicity of $\mathcal{DM}$*

It is trivially to prove that $\mathcal{DM}$ is monotone. None of the Datalog rules which define the direct mapping $\mathcal{DM}$ performs any alteration to the schema, the PKs, the FKs or the tuples of the instance I used as part of the input of $\mathcal{DM}$. Therefore, we conclude that $\mathcal{DM}$ is monotone.

D. Semantics preservation of $\mathcal{DM}$

The semantics preservation is more complicated then the rest of the properties. This is mainly due to monotonicity. First, we can give a counter-example and show that the direct mapping $\mathcal{DM}$ is not semantics preserving as the violation of integrity constraints for the primary and foreign keys are not reflected in the generated property graphs.

***Proposition** 1:* The direct mapping $\mathcal{DM}$ is not semantics preserving.

***Example** 3:* Assume that we have a relational schema containing a relation with name *Person* and attributes *name*,*DoB*, and assume that the attribute *name* is the primary key. Moreover, assume that this relation has two tuples, t_1 and t_2 such that $t_1.name = Bob, t_1.DoB = 1990, t_1.rId = 1$ and $t_2.name = Bob, t_2.DoB = 1991, t_2.rId = 2$. It is clear that the primary key is violated, therefore the database is inconsistent. However, we can see that the generated property graph is consistent after applying the mapping $\mathcal{DM}$. (The generated property graph can be seen in the Appendix)

We now show how to generate a semantics preserving direct mapping. In sub-section 1, we show how we can generate a semantics preserving direct mapping for primary keys by extending $\mathcal{DM}$. Then, in sub-section 2 we show how to deal with foreign keys. However, this is more difficult as monotonicity is a pitfall in this case as we know that no monotone direct mapping can be semantics preserving according to Sequeda et al. [11]

1) A semantics preserving direct mapping for primary keys: The integrity constraint for primary keys in a relational database says that a primary key is violated if there are repeated values or null vales. The defined mapping does not check if there exists primary keys with repeated values or null values in the instance I given as input. Therefore, in this case the mapping will generate regular nodes, properties and edges. This leads to an inconsistency between the relational database instance and the generated property graph. Therefore, we need a way to deal with this issue.

Consider a new direct mapping $\mathcal{DM}_{pk}$ that extends $\mathcal{DM}$ as follows. Two Datalog rules are used to determine if the value of a primary key attribute is repeated or if it is NULL. If some of these violations are found, then a node with an existing id will be generated that would produce an inconsistency as the mapping $\mathcal{DM}_{pk}$ will contain two nodes with the same id, which is inconsistent by the definition of $\mathcal{DM}$. Therefore, rule 14 from Table III is used to check if primary keys contain duplicates and rule 15 from Table III checks if primary keys contain null values.

***Proposition** 2:* The direct mapping DM_{pk} is information preserving, query preserving, monotone, and semantics preserving if one considers only PKs. That is, for every relational schema $\mathbf{R}$, set Σ of (only) PKs over $\mathbf{R}$ and instance I of $\mathbf{R}$: $I \models \Sigma$ iff $\mathcal{DM}_{pk}(\mathbf{R}, \Sigma, I)$ is consistent.

Information preservation, query preservation and monotonicity of $\mathcal{DM}_{pk}$ are corollaries of the fact that these properties hold for $\mathcal{DM}$, and of the fact that the Datalog rules introduced to handle primary keys are monotone.

Now, we analyze if $\mathcal{DM}_{pk}$ can also deal with foreign keys. However, this is not the case as the foreign keys constraint of having the same values is not checked during the mapping. Hence, the generated property graph is consistent.

2) A semantics preserving direct mapping for primary keys and foreign keys: Based on the theorem presented by Sequeda et al., no monotone direct mapping is semantics preserving. This is mainly because an inconsistency in a relational database which is reflected in a property graph can become consistent if new data is added to the database. In this case, the property graph has to be modified by removing elements. Hence, the desirable property of monotonicity does not hold.

We need to reflect any inconsistency from the relational database by creating an inconsistency in the property graph as well.Therefore, in this section, we present a non-monotone direct mapping $\mathcal{DM}_{pk+fk}$, which extends $\mathcal{DM}_{pk}$ by introducing new Datalog rules for verifying if there is a violation of a foreign key constraint. If such a violation exists, then a new node which already exists in the property graph will be generated by creating a node with an already existing id. More precisely, rule 16 from Table III checks if the attributes composing the foreign keys have the same values in the two relations. The predicate $IsVALUE_n$, defined by rule 17 in Table III, is used to check whether a tuple in a relation has values for some given attributes.

The inconsistency caused by a foreign key in a relational database may not be preserved if new data is added to the database. In this case the artificial nodes created by the previous rules must be removed. Therefore, the direct mapping $\mathcal{DM}_{pk+fk}$ is not monotone.

***Theorem** 3:* The direct mapping $\mathcal{DM}_{pk+fk}$ is information preserving, query preserving and semantics preserving.

Information preservation and query preservation of $\mathcal{DM}_{pk+fk}$ are corollaries of the fact that these properties hold for $\mathcal{DM}$ and $\mathcal{DM}_{pk}$.

VII. Engineering $\mathcal{DM}$

In the second part of the project, the described direct mapping has been implemented as well. The project is open-source and can be found in the following Github repository: `https://github.com/radualex/R2PG-DM`. The direct mapping has been implemented as a Java application using the build automation tool called Maven. [10] In order to construct the mapping, the JDBC library has been used for obtaining the schema of the database. We wanted the Java application to be as generic as possible. Thus, we would like to be able to connect any relational database management system (RDBMS) (i.e. Postgres, MySQL, Oracle, etc.) supported by JDBC driver as well. According to the official Oracle website, the following RDBMSs are supported by JDBC library: (1) MySQL 5.1, (2) Java DB 10.5.3.0, (3) Oracle 11, (4) PostgreSQL 8.4, (5) DB2 9.7, (6) Sybase ASE 15 and (7) Microsoft SQL Server 2008. [7] The Java application takes as input a database instance from which the metadata is extracted such as relation names and foreign keys. The output is stored into a relational instance as well

TABLE III

DATALOG RULES FOR $\mathcal{DM}_{pk}$ AND $\mathcal{DM}_{pk+fk}$

Rule #	Head	Body
14	$NODE(id(t.rId, R), R)$	$PK_n(X_1, ..., X_n, R), VALUE(V_1, X_1, t, R), ..., VALUE(V_n, X_n, t, R), V_1 = V_2 \vee V_1 = V_3 \vee ... \vee V_{n-2} = V_n \vee V_{n-1} = V_n$
15	$NODE(id(t.rId, R), R)$	$PK_n(X_1, ..., X_n, R), VALUE(V_1, X_1, t, R), ..., VALUE(V_n, X_n, t, R), V_1 = NULL \vee ... \vee V_n = NULL$
16	$NODE(id(t.rId, R), R)$	$FK_n(X_1, ..., X_n, R, Y_1, ..., Y_n, S), VALUE(V_1, X_1, t, R), ..., VALUE(V_n, X_n, t, R), V_1 \neq NULL, ..., V_n \neq NULL, \neg IsVALUE_n(V_1, ..., V_n, Y_1, ..., Y_n, S)$
17	$IsVALUE_n(V_1, ..., V_n, Y_1, ..., Y_n, S)$	$VALUE(V_1, Y_1, t, S), ..., VALUE(V_n, Y_n, t, S)$

into three different tables: $node$, $property$ and $edge$. The two connections are retrieved at run-time from a configuration file. It is important to mention that three classes are used for the implementation, each representing a property graph object as follows: (1) a $Node$ class is used for storing the node properties (id, ,label), (2) a $Property$ class is used for storing the property properties (id, key, value), and (3) an $Edge$ class is used for storing the edge properties (id, sourceId, targetId, label). The implementation starts by retrieving all the relational names from the input database. Then the algorithm is divided into two steps: (1) first, we create nodes and properties and (2) second, we create the edges.

In order to generate ids for nodes and edges, a function id with seven optional parameters as described in section V-B is used. The id will increment a global integer value by one with every call. The global integer starts at value 1.

A. Nodes and properties generation

The implementation for generating nodes and properties is quite straightforward. We iterate over the tables and for each relation name r we retrieve all the data, including the rId from the input database. This operation is a streaming process as we want to prevent an out of memory exception. Hence, we process several rows at a time in memory using the JDBC library. Finally, we call the id function which retrieves an unique id with the parameters: rId and relation name r, and we create a $node$ object for the relation name r and a $property$ object for each attribute of relation name r. Then, the data is inserted into the output database.

B. Edge generation

The implementation for edges is more complex than the nodes and properties. This is due to the fact that we need to retrieve the source and target nodes ids. For generating edges, we use the (compound) foreign keys as parameter. Therefore, for each (compound) foreign key that we find between a relation name r and a relation name s, we join the two relations based on the attributes of the (compound) foreign key in order to retrieve the values from the input database. Next, we use the attributes and values in a second query in order to retrieve the source and target nodes ids from the output database by querying the already created $node$ and $property$ relations. The result of the second query is an array of source or target nodes ids. Now having all the required data, an $Edge$ object is created and the data is inserted into the output database. This is the main idea behind our implementation.

The challenge of the implementation was to make the direct mapping scalable. We tried to avoid storing all data in memory and only then performing all the operations, as this would increase the risk of running out of memory for large databases. Therefore the scope of the operations involved in a translation has been reduced to a single table. After all operations are executed, the data is released from memory and we iterate to the next table. Thus, we tried to use the database as much as possible.

Now that we have the data, we would like to visualize the graph as well. Therefore, we have used Neo4j graph database for this matter. At the end of the mapping, we export the property graph data to $.csv$ files which, later, can be imported into Neo4j by using a custom cypher script file, which can be found in the repository as well. We will not show how to setup a Neo4j graph database in this paper as this is out of the scope of our topic. However, more information about this can be found in the GitHub repository. Figure 3 shows the running instance presented in figure 1 as a property graph in a Neo4j graph database.

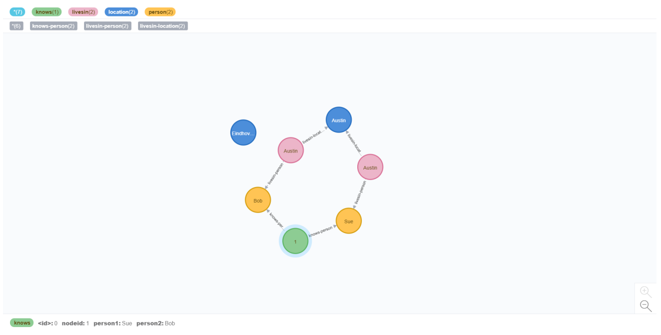


Fig. 3. Example Neo4j property graph for running instance

We tested the application with a database containing the running instance presented in figure 1 running in a PostgreSQL instance.

VIII. EXPERIMENT

After finishing the implementation, we have conducted a sequence of mappings for several databases from The CTU Prague Relational Learning Repository. [5] We tried to have as much variation as possible. Therefore, we have chosen

TABLE IV
DETAILS OF EXPERIMENT DATABASES

	Facebook	Basketball Women	Classic Models	Genes	Hepatitis	IMDb	Northwind	World	CORA
Rows	10.173	5.567	3.864	6.118	12.927	4.395	3.308	5.302	57.353
Attributes	265	176	59	15	26	22	191	24	6
Foreign keys	2	8	8	3	6	6	13	2	3
Tables	2	8	8	3	7	7	29	3	3
Nodes	10.173	5.567	3.864	6.118	12.927	4.395	3.308	5.302	57.353
Properties	163.113	172.976	20.723	44.478	78.105	16.889	24.296	27.783	114.706
Edges	19.252	53.777	6.846	6.166	13.016	4.448	7.113	1.216	60.074

eight different tables consisting of real and synthetic data and different tables structure such that we have databases with low number of tables and high number of attributes or databases with 'balanced' tables that have about the same number of attributes overall. The reason was to test our mapping and observe its behavior in different scenarios.

The databases chosen for this experiments are of different sizes and from different domains such as medical, financial or entertainment and some of them are also temporal: Facebook - a dataset consisting of real data of 'circles' (or 'friends lists') from Facebook; BasketballWomen - a dataset consisting of real data which predicts wheter a team plays playoff, or not; ClassicModels - a dataset consisting of synthetic data which stores data for a retailer of scale models of classic cars; Genes - a dataset consisting of real data which predicts the gene/protein function or localization; Hepatitis - a dataset consisting of real data contrasting cases of hepatitis type C against hepatitis type B; IMDb - a dataset consisting of real database of movies; Northwind - a dataset consisting of synthetic sales data of a fictional company called Northwind Traders, which imports and exports specialty foods from around the world; World - a dataset of real data consisting of real data countries, cities and languages spoken in different countries. This dataset is obtained from MySQL website. [6] And finally, CORA - a dataset of real data consisting of scientific publications. More details about these datasets are presented in table IV.

In order to check the correctness of the mapping, we can use some counting methods in the input database. For nodes, is trivial to see that the number of nodes should be equal to the number of rows for each database. For properties, we calculated for each relation name in the database the Cartesian product between the number of attributes and number of rows from which we subtract the null values. In order to get the expected number of properties, we sum all the results from all tables. Finally, for edges, we check for each foreign key how many rows have the same values for the attributes composing the foreign key. Then, we sum all the values from all foreign keys in the database in order to get the expected number of edges.

The execution of the mappings is in the range of 2.7 minutes for the world database and 87.3 minutes for the CORA database. We observed during the experiment a trend in the execution time. The databases with few tables and many attributes such as Facebook or BasketballWomen are much slower then the rest. We also observed that databases with balanced number of attributes over the tables have a better execution time. A possible issue for this is the lack of asynchronous behavior of the jdbc library. Currently Oracle are working on a new Asynchronous Database Access API. [8]. Other issues were encountered during the experiment as well. All databases involved in the experiment were in MySQL, which required several changes in the SQL statements from the initial implementation. The initial implementation was tested with PostgreSQL which contains non-standard SQL elements and it is also more flexible when parsing SQL queries. Therefore, when switching to MySQL, we have observed that some changes were required and the SQL queries were generalized such that they can be used in any RDBMS. Also, another aspect that slightly improved the performance was using the batch operation supported by the JDBC library which allows processing of multiple rows at a time instead of processing row by row.

For the future, improvements can be made for boosting the performance of the implementation by switching to the asynchronous JDBC API when it will be available or by creating a multi-threading environment in order to execute more SQL queries at the same time.

The experiment was conducted on a system with the following specifications: Intel$^{®}$ Core^{TM} i7-4510U CPU @ 2.00 Ghz, 8GB RAM. Also, we have tested with two different RDBMS's: PostgreSQL and MySQL.

IX. CONCLUDING REMARKS

To conclude the paper, we give a brief overview of the content. The main topic of this paper is how to map relational databases to property graphs. First, we analyze other direct mapping on the same topic and we emphasize on the issues of those approaches. Next, we present a direct mapping from relational databases to property graphs based on the direct mapping from relational databases to RDF graphs with OWL vocabulary of J. Sequeda. [11] We analyze two fundamental properties (information preservation and query preservation) and two desireable properties (monotonicity and semantics preservation). We prove the fundamental properties and we define an extended direct mapping for reflecting inconsistencies in databases, which is non-monotone and semantics preserving. Finally, we present an open-source project consisting of an implementation of the presented direct mapping in Java using JDBC library and Maven and an experiment where we map and analyze 8 different databases in order to check the results and the correctness of the implemented mapping.

Future work: We would like to investigate the PG Schema issue in more depth as the PG Schema standardization is still an open issue. We would also like to extend the schema to support properties for edges as well in order to take full advantage of the data model. Finally, we would like to create a customized mapping for cases where a schema is defined on the target instance.

ACKNOWLEDGMENT

The author would like to thank the supervisors of this project: George Fletcher and Juan Sequeda for all the support, time and feedback provided during the whole period of this master thesis.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432, 2018.

[3] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Morgan & Claypool, 2018.

[4] The PostgreSQL Global Development Group. *Chapter 8. Data Types. Section 8.18 Object Identifier Types*, 2018 (accessed May 30, 2019).

[5] Jan Motl and Oliver Schulte. The CTU prague relational learning repository. *CoRR*, abs/1511.03086, 2015.

[6] MySQL. Other mysql documentation. https://dev.mysql.com/doc/index-other.html, Last accessed: 2019-06-18.

[7] Oracle. Supported jdbc drivers and databases. https://docs.oracle.com/cd/E19226-01/820-7688/gawms/index.html, Last accessed: 2019-06-13.

[8] Oracle. Asynchronous database access api (adba), 2018. https://blogs.oracle.com/java/jdbc-next:-a-new-asynchronous-api-for-connecting-to-a-database, Last accessed: 2019-06-18.

[9] Ognjen Orel, Slaven Zakoek, and Mirta Baranovic. Property oriented relational-to-graph database conversion. *Automatika*, 57, 02 2017.

[10] Apache Maven Project. Welcome to apache maven. https://maven.apache.org, Last accessed: 2019-06-13.

[11] J. F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. On directly mapping relational databases to rdf and owl. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 649–658, New York, NY, USA, 2012. ACM.

[12] Radu Stoica, George H. L. 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, June 3-7, 2019.*, 2019.

[13] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. In *GRADES*, page 1. CWI/ACM, 2013.

[14] Kang Min Yoo, Sungchan Park, and Sang-goo Lee. Rdb2graph: A generic framework for modeling relational databases as graphs. In *JIST (Workshops & Posters)*, volume 1312 of *CEUR Workshop Proceedings*, pages 148–151. CEUR-WS.org, 2014.

Appendix I
EXAMPLES

A. Encoding of the running instance to Datalog rules

REL:

- $REL(''Person'')$: It identifies that Person is a relation in **R**.
- $REL(''Knows'')$: It identifies that Knows is a relation in **R**.
- $REL(''Location'')$: It identifies that Location is a relation in **R**.
- $REL(''LivesIn'')$: It identifies that LivesIn is a relation in **R**.

ATTR:

- $ATTR(''name'', ''Person'')$: Indicates that name is an attribute in the relation Person in **R**.
- $ATTR(''DoB'', ''Person'')$: Indicates that DoB is an attribute in the relation Person in **R**.
- $ATTR(''person1'', ''Knows'')$: Indicates that person1 is an attribute in the relation Knows in **R**.
- $ATTR(''person2'', ''Knows'')$: Indicates that person2 is an attribute in the relation Knows in **R**.
- $ATTR(''placename'', ''Location'')$: Indicates that placename is an attribute in the relation Location in **R**.
- $ATTR(''size'', ''Location'')$: Indicates that size is an attribute in the relation Location in **R**.
- $ATTR(''country'', ''Location'')$: Indicates that country is an attribute in the relation Location in **R**
- $ATTR(''name'', ''LivesIn'')$: Indicates that name is an attribute in the relation LivesIn in **R**.
- $ATTR(''placename'', ''LivesIn'')$: Indicates that placename is an attribute in the relation LivesIn in **R**.
- $ATTR(''country'', ''LivesIn'')$: Indicates that country is an attribute in the relation LivesIn in **R**

PK:

- $PK_1(''name'', ''Person'')$: Indicates that $Person[name]$ is a primary key in Σ.
- $PK_2(''placename'', ''country'', ''Location'')$: Indicates that $Location[placename, country]$ is a primary key in Σ.
- $PK_2(''person1'', ''person2'', ''Knows'')$: Indicates that $Knows[person1, person2]$ is a primary key in Σ.
- $PK_3(''name'', ''placename'', ''country'', ''LivesIn'')$: Indicates that $LivesIn[name, placename, country]$ is a primary key in Σ.

FK:

- $FK_1(''name'', ''Person'', ''name'', ''LivesIn'')$: Indicates that $Person[name] \subset LivesIn[name]$ is a foreign key in Σ.
- $FK_2(''placename'', ''country'', ''Location'', ''placename'', ''country'', ''LivesIn'')$: Indicates that $Location[placename, country] \subset LivesIn[placename, country]$ is a foreign key in Σ.
- $FK_1(''name'', ''Person'', ''person1'', ''Knows'')$: Indicates that $Person[name] \subset Knows[person1]$ is a foreign key in Σ.
- $FK_1(''name'', ''Person'', ''person2'', ''Knows'')$: Indicates that $Person[name] \subset Knows[person2]$ is a foreign key in Σ.

VALUE:

- $VALUE(''Bob'', ''name'', ''1'', ''Person'')$: Indicates that Bob is the value of an attribute name in a tuple with identifier 1 in a relation Person from **R**.
- $VALUE(''Sue'', ''name'', ''2'', ''Person'')$: Indicates that Sue is the value of an attribute name in a tuple with identifier 2 in a relation Person from **R**.
- $VALUE(''1990'', ''DoB'', ''1'', ''Person'')$: Indicates that 1990 is the value of an attribute DoB in a tuple with identifier 1 in a relation Person from **R**.
- $VALUE(''NULL'', ''DoB'', ''2'', ''Person'')$: Indicates that NULL is the value of an attribute DoB in a tuple with identifier 2 in a relation Person from **R**.
- $VALUE(''Sue'', ''person1'', ''1'', ''Knows'')$: Indicates that Sue is the value of an attribute person1 in a tuple with identifier 1 in a relation Knows from **R**.
- $VALUE(''Bob'', ''person2'', ''1'', ''Knows'')$: Indicates that Bob is the value of an attribute person2 in a tuple with identifier 1 in a relation Knows from **R**.
- $VALUE(''Austin'', ''placename'', ''1'', ''Location'')$: Indicates that Austin is the value of an attribute placename in a tuple with identifier 1 in a relation Location from **R**.
- $VALUE(''Eindhoven'', ''placename'', ''2'', ''Location'')$: Indicates that Eindhoven is the value of an attribute placename in a tuple with identifier 2 in a relation Location from **R**.
- $VALUE(''big'', ''size'', ''1'', ''Location'')$: Indicates that big is the value of an attribute size in a tuple with identifier 1 in a relation Location from **R**.
- $VALUE(''small'', ''size'', ''2'', ''Location'')$: Indicates that small is the value of an attribute size in a tuple with identifier 2 in a relation Location from **R**.

- $VALUE(”USA”,”country”,”1”,”Location”)$: Indicates that USA is the value of an attribute country in a tuple with identifier 1 in a relation Location from **R**.
- $VALUE(”NL”,”country”,”2”,”Location”)$: Indicates that NL is the value of an attribute country in a tuple with identifier 2 in a relation Location from **R**.
- $VALUE(”Bob”,”name”,”1”,”LivesIn”)$: Indicates that Bob is the value of an attribute name in a tuple with identifier 1 in a relation LivesIn from **R**.
- $VALUE(”Sue”,”name”,”2”,”LivesIn”)$: Indicates that Sue is the value of an attribute name in a tuple with identifier 2 in a relation LivesIn from **R**.
- $VALUE(”Austin”,”placename”,”1”,”LivesIn”)$: Indicates that Austin is the value of an attribute placename in a tuple with identifier 1 in a relation LivesIn from **R**.
- $VALUE(”Austin”,”placename”,”2”,”LivesIn”)$: Indicates that Austin is the value of an attribute placename in a tuple with identifier 2 in a relation LivesIn from **R**.
- $VALUE(”USA”,”country”,”1”,”LivesIn”)$: Indicates that USA is the value of an attribute country in a tuple with identifier 1 in a relation LivesIn from **R**.
- $VALUE(”USA”,”country”,”2”,”LivesIn”)$: Indicates that USA is the value of an attribute country in a tuple with identifier 2 in a relation LivesIn from **R**.

B. Translation of the running instance to property graph

Nodes:

- $NODE(”1”,”Person”) \leftarrow REL(”Person”), VALUE(”Bob”,”name”,”1”,”Person”), VALUE(”1980”,”DoB”,”1”,”Person”)$
- $NODE(”2”,”Person”) \leftarrow REL(”Person”), VALUE(”Sue”,”name”,”2”,”Person”)$
- $NODE(”3”,”Knows”) \leftarrow REL(”Knows”), VALUE(”Sue”,”person1”,”1”,”Knows”), VALUE(”Bob”,”person2”,”1”,”Knows”)$
- $NODE(”4”,”Location”) \leftarrow REL(”Location”), VALUE(”Austin”,”placename”,”1”,”Location”), VALUE(”big”,”size”,”1”,”Location”), VALUE(”USA”,”country”,”1”,”Location”)$
- $NODE(”5”,”Location”) \leftarrow REL(”Location”), VALUE(”Eindhoven”,”placename”,”2”,”Location”), VALUE(”small”,”size”,”2”,”Location”), VALUE(”NL”,”country”,”2”,”Location”)$
- $NODE(”6”,”LivesIn”) \leftarrow REL(”LivesIn”), VALUE(”Bob”,”name”,”1”,”LivesIn”), VALUE(”Austin”,”placename”,”1”,”LivesIn”), VALUE(”USA”,”country”,”1”,”LivesIn”)$
- $NODE(”7”,”LivesIn”) \leftarrow REL(”LivesIn”), VALUE(”Sue”,”name”,”2”,”LivesIn”), VALUE(”Austin”,”placename”,”2”,”LivesIn”), VALUE(”USA”,”country”,”2”,”LivesIn”)$

Edges:

- $EDGE_1(”8”,”3”,”2”,”Knows-Person”) \leftarrow FK_1(”name”,”Person”,”person1”,”Knows”), VALUE(”Sue”,”name”,”2”,”Person”), VALUE(”Sue”,”person1”,”1”,”Knows”), CONCAT_3(”Knows”,”-”,”Person”,”Knows-Person”)$
- $EDGE_1(”9”,”3”,”1”,”Knows-Person”) \leftarrow FK_1(”name”,”Person”,”person2”,”Knows”), VALUE(”Bob”,”name”,”1”,”Person”), VALUE(”Bob”,”person2”,”1”,”Knows”), CONCAT_3(”Knows”,”-”,”Person”,”Knows-Person”)$
- $EDGE_1(”10”,”6”,”1”,”LivesIn-Person”) \leftarrow FK_1(”name”,”Person”,”name”,”LivesIn”), VALUE(”Bob”,”name”,”1”,”Person”), VALUE(”Bob”,”name”,”1”,”LivesIn”), CONCAT_3(”LivesIn”,”-”,”Person”,”LivesIn-Person”)$
- $EDGE_1(”11”,”7”,”2”,”LivesIn-Person”) \leftarrow FK_1(”name”,”Person”,”name”,”LivesIn”), VALUE(”Sue”,”name”,”2”,”Person”), VALUE(”Sue”,”name”,”2”,”LivesIn”), CONCAT_3(”LivesIn”,”-”,”Person”,”LivesIn-Person”)$
- $EDGE_1(”12”,”6”,”4”,”LivesIn-Location”) \leftarrow FK_2(”placename”,”country”,”Location”,”placename”,”country”,”LivesIn”), VALUE(”Austin”,”placename”,”1”,”Location”), VALUE(”USA”,”country”,”1”,”Location”), VALUE(”Austin”,”placename”,”1”,”LivesIn”), VALUE(”USA”,”country”,”1”,”LivesIn”), CONCAT_3(”LivesIn”,”-”,”Location”,”LivesIn-Location”)$
- $EDGE_1(”13”,”7”,”4”,”LivesIn-Location”) \leftarrow FK_2(”placename”,”country”,”Location”,”placename”,”country”,”LivesIn”), VALUE(”Austin”,”placename”,”1”,”Location”), VALUE(”USA”,”country”,”1”,”Location”), VALUE(”Austin”,”placename”,”2”,”LivesIn”), VALUE(”USA”,”country”,”2”,”LivesIn”), CONCAT_3(”LivesIn”,”-”,”Location”,”LivesIn-Location”)$

Properties:

- $PROPERTY(”1”,”name”,”Bob”) \leftarrow NODE(”1”,”Person”), VALUE(”Bob”,”name”,”1”,”Person”)$
- $PROPERTY(”1”,”DoB”,”1990”) \leftarrow NODE(”1”,”Person”), VALUE(”1990”,”DoB”,”1”,”Person”)$

- $PROPERTY("2","name","Sue") \leftarrow NODE("2","Person"), VALUE("Sue","name","2","Person")$
- $PROPERTY("3","person1","Sue") \leftarrow NODE("3","Knows"), VALUE("Sue","person1","1","Knows")$
- $PROPERTY("3","person2","Bob") \leftarrow NODE("3","Knows"), VALUE("Bob","person2","1","Knows")$
- $PROPERTY("4","placename","Austin") \leftarrow NODE("4","Location"), VALUE("Austin","placename","1","Location")$
- $PROPERTY("4","size","big") \leftarrow NODE("4","Location"), VALUE("big","size","1","Location")$
- $PROPERTY("4","country","USA") \leftarrow NODE("4","Location"), VALUE("USA","country","1","Location")$
- $PROPERTY("5","placename","Eindhoven") \leftarrow NODE("5","Location"), VALUE("Eindhoven","placename","2","Location")$
- $PROPERTY("5","size","small") \leftarrow NODE("5","Location"), VALUE("small","size","2","Location")$
- $PROPERTY("5","country","NL") \leftarrow NODE("5","Location"), VALUE("NL","country","2","Location")$
- $PROPERTY("6","name","Bob") \leftarrow NODE("6","LivesIn"), VALUE("Bob","name","1","LivesIn")$
- $PROPERTY("6","placename","Austin") \leftarrow NODE("6","LivesIn"), VALUE("Austin","placename","1","LivesIn")$
- $PROPERTY("6","country","USA") \leftarrow NODE("6","LivesIn"), VALUE("USA","country","1","LivesIn")$
- $PROPERTY("7","name","Sue") \leftarrow NODE("7","LivesIn"), VALUE("Sue","name","2","LivesIn")$
- $PROPERTY("7","placename","Austin") \leftarrow NODE("7","LivesIn"), VALUE("Austin","placename","2","LivesIn")$
- $PROPERTY("7","country","USA") \leftarrow NODE("7","LivesIn"), VALUE("USA","country","2","LivesIn")$

Appendix II PROOFS

A. Proof of Theorem 1

Proof:

We show that $\mathcal{DM}$ is information preserving by providing a computable mapping $\mathcal{N} : \mathcal{G} \to I$ that satisfies the condition in Definition 3. More precisely, given a relation schema $\mathbf{R}$, a set Σ of PKs and FKs and an instance I of $\mathbf{R}$ satisfying Σ, next we show how $\mathcal{N}(\mathcal{G})$ is defined for $\mathcal{DM}(\mathbf{R}, \Sigma, I) = \mathcal{G}$.

- **Step 1:** In order to retrieve all the row ids from the mapping obtained with the function id, we need an inverse function, $id^{-1} : \mathbb{N} \to (\mathbb{N} \cup \{NULL\}) \times (\mathbf{R} \cup \{NULL\}) \times (\mathcal{A} \cup \{NULL\}) \times (\mathbb{N} \cup \{NULL\}) \times (\mathbf{R} \cup \{NULL\}) \times \bigcup_{i \geq 0}(\mathcal{A} \cup \{NULL\} \times \mathcal{A} \cup \{NULL\})$. By using the id^{-1} function and the output of an id function, we are able to retrieve the row id for a tuple.
- **Step 2:** Identify all the nodes with label "Rel" from the schema (i.e. $NODE(id(r'),"Rel")$). The function $id(r')$ identifies a relation name r' from $\mathcal{G}$. For every r' that was retrieved from $\mathcal{G}$, map it to a relation name r.
- **Step 3:** Identify all the nodes with label "Att" from the schema (i.e. $NODE(id(r', a'),"Att")$. The function $id(r', a')$ identifies an attribute a' in a relation name r' from $\mathcal{G}$. Every attribute a' in r' is mapped to an attribute a in r.
- **Step 4:** Identify all the primary keys using the $PROPERTY$ table where we store the primary keys for each relation name in $\mathbf{R}$. The rule $\bigcup_{n=0}^{\overline{R}} (\bigcup_{i=1}^{n} PROPERTY_n(id(r_n, a_i),"pk","true") \leftarrow PK_n(a_1, ..., a_i, ..., a_n, r_n))$ is used for this purpose.
- **Step 5:** Identify all the nodes with label "Fk" from the relation schema. (i.e. $\bigcup_{n=0}^{\overline{FK}} NODE_n(id(r, s, a_1, ..., a_n, b_1, ..., b_n),"Fk") \leftarrow FK_n(a_1, ..., a_n, r, b_1, ..., b_n, s)))$. The relation names r' and s' from G are mapped to the relation names r and s. The attributes $a'_1, ..., a'_n$ over r' are mapped to attributes $a_1, ..., a_n$ over r and the attributes $b'_1, ..., b'_n$ over s' are mapped to attributes $b_1, ..., b_n$ over s. The data represents a foreign key $r[a_1, ..., a_n] \subseteq s[b_1, ..., b_n]$.
- **Step 6:** Identify all non-NULL tuples from G using the following Datalog rule over the property graph rules:
$$VALUE(v, a, t.rId, r) \leftarrow NODE(id(t.rId, r), r), PROPERTY(id(t.rId, r), a, v), id^{-1}(id(t.rId, r)).$$
The rule $id^{-1}(id(t.rId, r))$ is used to retrieve the identifier rId for a tuple t.
- **Step 7:** In order to completely retrieve the whole information, we need to consider the attributes which were NULL in the input instance I, as these attributes are not reflected in the generated property graph. Therefore, we will use the following rule in order to obtain the NULL values by mixing elements from property graph and the schema as follows:
$$\begin{gathered} VALUE(NULL, a, t.rId, r) \leftarrow \\ NODE(id(t.rId, r), r), PROPERTY(id(t.rId, r), a_1, v_1), NODE(id(r),"Rel"), NODE(id(r, a_1),"Att"), \\ NODE(id(r, a),"Att"), EDGE(id(id(r), id(r, a)), id(r), id(r, a),"Rel-Att") \end{gathered}$$

We will also make the assumption that the relation names which contain attributes with NULL values have at least two attributes, otherwise the database will be inconsistent. Assume that relation name r has only one attribute a, which is also, by default, the primary key of relation name r. Therefore, if a is NULL for a tuple t, then the primary key for the tuple t will also be NULL, which implies the inconsistency. Therefore, if r has an attribute with a NULL value, the relation name r must have at least two attributes.

It is straightforward to prove that for every relational schema $\mathbf{R}$, set Σ of PKs and FKs and an instance I of $\mathbf{R}$ satisfying Σ, it holds that $\mathcal{N}(\mathcal{M}(\mathbf{R}, \Sigma, I)) = I$. This concludes the proof of the theorem. ■

B. Proof of Theorem 2

Proof: We need to prove that for every relatioal schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$, and relational algebra query Q over $\mathbf{R}$, there exists a G-CORE query Q^* such that for every instance I of $\mathbf{R}$ including null values:

$$tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$$

The following lemma is used in the proof of the theorem.

***Lemma** 1:* Let Q_1 be a relational algebra query over $\mathbf{R}$ and $R \in \mathbf{R}$ a relation name such that $att(Q_1) = \{A_1, ..., A_l\}$, and assume that Q_1^* is a G-CORE query such that:

$$tr(\llbracket Q_1 \rrbracket_I) = \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$$

Let

$$\begin{aligned} \nu = \ & \text{SELECT } n.A_1, ..., n.A_l \\ & \text{MATCH}(n : R) \\ & \text{ON } \mathcal{G} \\ & \text{OPTIONAL } (Q_1^*) \end{aligned}$$

Then we have that:

$$tr(\llbracket Q_1 \rrbracket_I) = \llbracket \nu \rrbracket_{\Omega,\mathcal{G}}$$

Proof: First, we prove that $tr(\llbracket Q_1 \rrbracket_I) \subseteq \llbracket \nu \rrbracket_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr(\llbracket Q_1 \rrbracket_I)$. Then there exists a tuple $t \in \llbracket Q_1 \rrbracket_I$ such that $tr(t) = \mu$. Thus, given that $att(Q_1) = \{A_1, ..., A_l\}$, we conclude that $dom(\mu) \subseteq \{A_1, ..., A_l\}$. Given that $tr(\llbracket Q_1 \rrbracket_I) = \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$, we have that $\mu \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$. Hence, from the fact that $dom(\mu) \subseteq \{A_1, ..., A_l\}$, we conclude that $\mu \in \llbracket \nu \rrbracket_{\Omega,\mathcal{G}}$.

Second, we prove that $\llbracket \nu \rrbracket_{\Omega,\mathcal{G}} \subseteq tr(\llbracket Q_1 \rrbracket_I)$. Assume that $\mu \in \llbracket \nu \rrbracket_{\Omega,\mathcal{G}}$. Then there exists a mapping $\mu' \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$ such that $\mu = \mu'_{|\{A_1,...,A_l\}}$. From the fact that $tr(\llbracket Q_1 \rrbracket_I) = \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$, we conclude that $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$. Thus, there exists a tuple $t \in \llbracket Q_1 \rrbracket_I$ such that $tr(t) = \mu'$. But then given that $att(Q_1) = \{A_1, ..., A_l\}$, we conclude by definition of tr that $dom(\mu') \subseteq \{A_1, ..., A_l\}$. Therefore, given that $\mu = \mu'_{|\{A_1,...,A_l\}}$, we have that $\mu = \mu'$ and, hence, $\mu \in tr(\llbracket Q_1 \rrbracket_I)$ since $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$. ■

We now prove the theorem by induction on the structure of relational algebra query Q. Let $\mathcal{G}$ be the property graph resulted from the mapping $\mathcal{DM}(\mathbf{R}, \Sigma, I)$.

Base case: In this part of the proof, we need to consider two cases.

- **Non-empty relations:** Assume that Q is the identity relational algebra query R, where R is a non-binary relation as defined in this section. Moreover, assume that $att(R) = \{A_1, ..., A_l\}$. Finally, let Q^* be the following G-CORE query:

$$\begin{aligned} & \text{SELECT } n.A_1, ..., n.A_l \\ & \text{MATCH}(n : R) \\ & \text{ON } \mathcal{G} \end{aligned}$$

  Next, we prove that $tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$.

  First, we show that $tr(\llbracket Q \rrbracket_I) \subseteq \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr(\llbracket Q \rrbracket_I)$. Then there exists a tuple $t \in \llbracket Q \rrbracket_I$ such that $tr(t) = \mu$ and, hence, $t \in R^I$. Without loss of generality, assume that there exists $k \in \{0, ..., l\}$ such that (1) $t.A_i \neq NULL$ for every $i \in \{1, ..., k\}$, and that $dom(\mu) = \{A_1, ..., A_l\}$. Therefore, given the definition of $\mathcal{DM}$ and the fact that R is not a binary relation, we have that the following property graph elements are included in $\mathcal{DM}(\mathbf{R}, \Sigma, I)$:

  - $NODE(id(t.rId, R), R)$, where $t.rId$ is the tuple id for the tuple t, and
  - $PROPERTY(id(t.rId, R), A_i, V_i)$, where $i \in \{1, ..., k\}$ and V_i is the value of attribute A_i in the tuple t, that is, $t.A_i = V_i$.

  Thus, given that no element of the form $PROPERTY(id(t.rId, R), A_j, V_j)$ is included in $\mathcal{DM}(\mathbf{R}, \Sigma, I)$, for $j \in \{k+1, ..., l\}$, we conclude that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$ by definition of Q^* and the fact that $\mu = tr(t)$.

Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Without loss of generality, assume that $dom(\mu) = \{A_1, ..., A_l\}$, where $0 \leq k \leq l$. Then by definition of Q^*, we have that $\mathcal{DM}(\mathbf{R}, \Sigma, I)$ contains node $NODE(id(t.rId, R), R)$ and properties $PROPERTY(id(t.rId, R)$
$, A_i, \mu(A_i))$, for every $i \in \{1, ..., k\}$, and it does not contain properties of the form $PROPERTY$
$(id(t.rId, R), A_j, V_j)$, for every $j \in \{k+1, ..., l\}$. Given the definition of $\mathcal{DM}(\mathbf{R}, \Sigma, I)$ and the fact that R is not binary, we conclude that there exists a tuple $t \in R^I$ such that: (1) $t.A_i = \mu(A_i)$ for every $i \in \{1, ..., k\}$ and (2) $t.A_j = NULL$ for every $j \in \{k+1, ..., l\}$. Thus, given that $tr(t) = \mu$ and $t \in R^I$, we conclude that $\mu \in tr([\![Q]\!]_I)$. (recall that $[\![Q]\!]_I = R^I$).

- **Empty relations:** Assume that $Q = NULL_A$, and let Q^* be the G-CORE query $SELECT* \ MATCH() \ FROM$ $\mathcal{G}$. We have that $[\![Q]\!]_I = \{t\}$, where t is a tuple with domain $\{A\}$ such that $t.A = NULL$. Moreover, we have that $[\![Q^*]\!]_{\Omega,\mathcal{G}} = \{\mu_\emptyset\}$ since $\mathcal{G}$ is a nonempty property graph. Thus, given that $tr(t) = \mu_\emptyset$, we conclude that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.

Inductive step: Assume that the theorem holds for relational algebra queries Q_1 and Q_2. That is, there exists G-CORE queries Q_1^* and Q_2^* such that:

$$tr([\![Q_1]\!]_I) = [\![Q_1^*]\!]_{\Omega,\mathcal{G}},$$
$$tr([\![Q_2]\!]_I) = [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$$

To continue with the proof, we need to consider the following operators: selection (σ), projection (π), rename (δ), join ($\bowtie$), union ($\cup$) and difference ($\setminus$).

- **Selection**: We need to consider four cases.
  - **Case 1:** Assume that $Q = \sigma_{A_1=a}(Q_1)$, and

    Q^* = SELECT $n.A_1$
    MATCH($n : r$)
    ON $\mathcal{G}$
    OPTIONAL (Q_1^*)
    WHERE $n.A_1 = a$
    AND $a \neq NULL$

    Next, we prove that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
    First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then, there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ and $t.A_1 = a$. By definition of tr, we know that $t.A_1 = \mu(A_1)$, from which we conclude that $\mu(A_1) = a$ given that $t.A_1 = a$. Therefore, $\mu \models (n.A_1 = a$ AND $a \neq NULL)$, from which we conclude that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$ since $\mu = tr(t)$ and $tr(t) \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ by induction hypothesis.
    Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \models (n.A_1 = a$ AND $a \neq NULL)$, that is, $\mu(A_1) = a$. By induction hypothesis, we have that $\mu \in tr([\![Q_1]\!]_I)$, and, hence, there exists a tuple $t \in [\![Q_1]\!]_I$ such that $tr(t) = \mu$. By definition of tr, we know that $t.A_1 = \mu(A_1)$, from which we conclude that $t.A_1 = a$ given that $\mu(A_1) = a$. Given that $t \in [\![Q_1]\!]_I$ and $t.A_1 = a$, we have that $t \in [\![Q]\!]_I$. Therefore, we conclude that $\mu \in tr([\![Q]\!]_I)$ since $tr(t) = \mu$.
  - **Case 2:** Assume that $Q = \sigma_{A_1 \neq a}(Q_1)$, and

    Q^* = SELECT $n.A_1$
    MATCH($n : R$)
    ON $\mathcal{G}$
    OPTIONAL (Q_1^*)
    AND $n.A_1 \neq NULL$
    AND $n.A_1 \neq a$

    Next, we prove that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
    First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then, there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ and $t.A_1 \neq a$ and $t.A_1 \neq NULL$. By definition of tr, we know that $t.A_1 = \mu(A_1)$ and $\mu(A_1) \neq a$. Therefore, $\mu \models (n.A_1 \neq NULL$ AND $n.A_1 \neq a)$, from which we conclude that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$ since $\mu = tr(t)$ and $tr(t) \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$, by induction hypothesis.
    Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \models (n.A_1 \neq NULL$ AND $n.A_1 \neq a)$, that is, $A_1 \in dom(\mu)$ and $\mu(A_1) \neq a$. By induction hypothesis, we have that $\mu \in$

$tr([\![Q_1]\!]_I)$, and, hence, there exists a tuple $t \in [\![Q_1]\!]_I$ such that $tr(t) = \mu$. Given that $A_1 \in dom(\mu)$ and $\mu(A_1) \neq a$, it holds that $t.A_1 \neq NULL$ and $t.A_1 \neq a$. Thus, we have that $t \in [\![Q]\!]_I$, from which we conclude that $\mu \in tr([\![Q]\!]_I)$ since $\mu = tr(t)$.

- **Case 3:** Assume that $Q = \sigma_{IsNull(A_1)}(Q_1)$, and

  Q^* = SELECT $n.A_1$
  MATCH($n : R$)
  ON $\mathcal{G}$
  OPTIONAL (Q_1^*)
  AND $n.A_1 = NULL$

  Next, we prove that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
  First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then, there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ and $t.A_1 = NULL$. By definition of tr, we know that $A_1 \notin dom(\mu)$ and, hence, $\mu \models (n.A_1 = NULL)$. Therefore, we have that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$ given that $\mu = tr(t)$ and $tr(t) \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ by induction hypothesis.
  Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \models (n.A_1 = NULL)$, that is, $A_1 \notin dom(\mu)$. By induction hypothesis, we have that $\mu \in tr([\![Q_1]\!]_I)$, and, hence, there exists a tuple $t \in [\![Q_1]\!]_I$ such that $tr(t) = \mu$. Given that $A_1 \notin dom(\mu)$, it holds that $t.A_1 = NULL$. Thus, we have that $t \in [\![Q]\!]_I$, from which we conclude that $\mu \in tr([\![Q]\!]_I)$ since $\mu = tr(t)$.
- **Case 4:** Assume that $Q = \sigma_{IsNotNull(A_1)}(Q_1)$, and

  Q^* = SELECT $n.A_1$
  MATCH($n : R$)
  ON $\mathcal{G}$
  OPTIONAL (Q_1^*)
  AND $n.A_1 \neq NULL$

  Next, we prove that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
  First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then, there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ and $t.A_1 \neq NULL$. By definition of tr, we know that $A_1 \in dom(\mu)$ and, hence, $\mu \models (n.A_1 \neq NULL)$. Therefore, we have that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$ given that $\mu = tr(t)$ and $tr(t) \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ by induction hypothesis.
  Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \models (n.A_1 \neq NULL)$, that is, $A_1 \in dom(\mu)$. By induction hypothesis, we have that $\mu \in tr([\![Q_1]\!]_I)$, and, hence, there exists a tuple $t \in [\![Q_1]\!]_I$ such that $tr(t) = \mu$. Thus, be definition of tr we have that $t.A_1 = \mu(A_1)$, which implies that $t.A_1 \neq NULL$. Therefore, we have that $t \in [\![Q]\!]_I$ and, hence, $\mu \in tr([\![Q]\!]_I)$ since $\mu = tr(t)$.

- **Projection:** Assume that $Q = \pi_{A_1,...,A_l}(Q_1)$, and

  Q^* = SELECT $n.A_1, ..., n.A_l$
  MATCH($n : R$)
  ON $\mathcal{G}$
  OPTIONAL (Q_1^*)

  Next, we prove that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
  First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Given that $t \in [\![Q]\!]_I$, there exists a tuple $t' \in [\![Q_1]\!]_I$ such that for every $A \in att(Q) : t.A = t'.A$. Without loss of generality, assume that: (1) $att(Q) = \{A_1, ..., A_k, A_{k+1}, ..., A_l\}$, (2) $t.A_i \neq NULL$ for every $i \in \{1, ..., k\}$, and (3) $t.A_j = NULL$ for every $j \in \{k+1, ..., l\}$. By definition of tr, we have that $t.A_i = \mu(A_i)$ for every $i \in \{1, ..., k\}$, and that $dom(\mu) = \{A_1, ..., A_k\}$. Given that $t' \in [\![Q_1]\!]_I$, we have for $\mu' = tr(t')$ that: (1) $\mu' \in tr([\![Q_1]\!]_I)$, (2) $dom(\mu) \subseteq dom(\mu')$, (3) $(\mu) = (\{A_1, ..., A_l\} \cap dom(\mu'))$ and (4) $t.A_i = t'.A_i = \mu(A_i) = \mu'(A_i)$ for every $i \in \{1, ..., k\}$. Thus, we have in particular that:

  $$\mu = \mu'_{|\{A_1,...,A_l\}} \ (P_1)$$

  By induction hypothesis we have that $\mu' \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$, from which we conclude that $\mu'_{|A_1,...,A_l} \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Thus, we conclude from (P_1) that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
  Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then there exists a mapping $\mu' \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ such

that $\mu = \mu'_{|\{A_1,...,A_l\}}$. By induction hypothesis, we have that $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$, from which we conclude that there exists a tuple $t' \in \llbracket Q_1 \rrbracket_I$ such that $tr(t') = \mu'$. Let t be a tuple with domain $\{A_1, ..., A_l\}$ such that $t.A_i = t'.A_i$ for every $i \in \{1, ..., l\}$. Then, given that $t' \in \llbracket Q_1 \rrbracket_I$, we have that $t \in \llbracket Q \rrbracket_I$, and given that $\mu' = tr(t')$ and $\mu = \mu'_{|\{A_1,...,A_l\}}$, we have that $\mu = tr(t)$. Therefore, we conclude that $\mu \in tr(\llbracket Q \rrbracket_I)$.

- **Rename:** Assume that $att(Q) = \{A_1, ..., A_l\}$ and $Q = \delta_{A_1 \to B_1}(Q_1)$ and let

  Q^* = SELECT $n.A_1$ AS $B_1, ..., n.A_l$
  MATCH($n : R$)
  ON $\mathcal{G}$
  OPTIONAL (Q_1^*)

  Next, we prove that $tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$.
  First, we show that $tr(\llbracket Q \rrbracket_I) \subseteq \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr(\llbracket Q \rrbracket_I)$. Then there exists a tuple $t \in \llbracket Q \rrbracket_I$ such that $tr(t) = \mu$. Given that $t \in \llbracket Q \rrbracket_I$, there exists a tuple $t' \in \llbracket Q_1 \rrbracket_I$ such that $t.B_1 = t'.A_1$ and $t.A_i = t'.A_i$ for every $i \in \{2, ..., l\}$. Without loss of generality, assume that there exists $k \in \{1, ..., l\}$ such that: (1) $t.A_i \neq NULL$ for every $i \in \{2, ..., k\}$, and (2) $t.A_j = NULL$ for every $j \in \{k + 1, ..., l\}$. To finish the proof, we consider two cases.
  - Assume that $t.B_1 \neq NULL$. Then it follows from condition (1), (2) and definition of tr that $\mu(A_1) = t.B_1 = t'.A_1$, $\mu(A_i) = t.A_i = t'.A_i$ for every $i \in \{2, ..., k\}$ and $dom(\mu) = \{A_1, A_2, ..., A_k\}$. Let $\mu' = tr(t')$. Then by definition of tr, we have that $\rho_{\{A_1 \to B_1\}}(\mu') = \mu$. Moreover, given that $\mu' = tr(t')$ and $t' \in \llbracket Q_1 \rrbracket_I$, we conclude that $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$ and, hence, $\mu' \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$ by induction hypothesis. Thus, we have that $\rho_{\{A_1 \to B_1\}}(\mu') \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$, from which we conclude that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$ since $\mu = \mu'_{|\{A_1,...,A_l\}} = \mu'$ and $\rho_{\{A_1 \to B_1\}}(\mu') = \mu$.
  - Assume that $t.B_1 = NULL$. Then it follows from condition (1), (2) and definition of tr that $\mu(A_1) = t.A_1 = t'.A_i$ for every $i \in \{2, ..., k\}$ and $dom(\mu) = \{A_2, A_2, ..., A_k\}$. Let $\mu' = tr(t')$. Then by definition of tr, we have that $\rho_{\{A_1 \to B_1\}}(\mu') = \mu$. Moreover, given that $\mu' = tr(t')$ and $t' \in \llbracket Q_1 \rrbracket_I$, we conclude that $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$ and, hence, $\mu' \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$ by induction hypothesis. Thus, we have that $\rho_{\{A_1 \to B_1\}}(\mu'_{|\{A_1,...,A_l\}}) \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$, from which we conclude that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$ since $\mu'_{|\{A_1,...,A_l\}} = \mu'$ and $\rho_{\{A_1 \to B_1\}}(\mu') = \mu$.

  Second, we show that $\llbracket Q^* \rrbracket_{\Omega,\mathcal{G}} \subseteq tr(\llbracket Q \rrbracket_I)$. Assume that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Then there exists a mapping $\mu' \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$ such that $\mu = \rho_{\{A_1 \to B_1\}}(\mu'_{|\{A_1,...,A_l\}})$. By induction hypothesis, we have that $\mu' \in tr(\llbracket Q_1 \rrbracket_I)$, from which we conclude that there exists a tuple $t' \in \llbracket Q_1 \rrbracket_I$ such that $tr(t') = \mu'$. Let t be a tuple with domain $\{B_1, A_2, ..., A_l\}$ such that $t.B_1 = t'.A_1$ and $t.A_i = t'.A_i$ for every $i \in \{2, ..., l\}$. Then we have that $t \in \llbracket Q \rrbracket_I$. Given that $\mu' = tr(t')$ and $\mu = \rho_{\{A_1 \to B_1\}}(\mu'_{|\{A_1,...,A_l\}})$, we have that $\mu = tr(t)$. Therefore, we conclude that $\mu \in tr(\llbracket Q \rrbracket_I)$.
- **Join:** Assume that $Q = (Q_1 \bowtie Q_2)$, where $(att(Q_1) \cap att(Q_2)) = \{A_1, ..., A_l\}$, and let:

  Q^* = (Q_1^* INTERSECT Q_2^*)

  Next, we prove that $tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$.
  First, we show that $tr(\llbracket Q \rrbracket_I) \subseteq \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr(\llbracket Q \rrbracket_I)$. Then there exists a tuple t such that $\mu = tr(t)$ and $t \in \llbracket Q \rrbracket_I$. Thus, we have that there exist tuples $t_1 \in \llbracket Q_1 \rrbracket_I$ and $t_2 \in \llbracket Q_2 \rrbracket_I$ such that: (1) $t.A_i = t_1.A_i = t_2.A_i \neq NULL$ for every $i \in \{1, ..., l\}$, (2) $t.A = t_1.A$ for every $A \in (att(Q_1) \setminus att(Q_2))$, and (3) $t.A = t_2.A$ for every $A \in (att(Q_2) \setminus att(Q_1))$. Let $\mu_1 = tr(t_1)$ and $\mu_2 = tr(t_2)$. By induction hypothesis and given that $\mu_1 \in tr(\llbracket Q_1 \rrbracket_I)$ and $\mu_2 \in tr(\llbracket Q_2 \rrbracket_I)$, we have that $\mu_1 \in \llbracket Q_1^* \rrbracket_{\Omega,\mathcal{G}}$ and $\mu_2 \in \llbracket Q_2^* \rrbracket_{\Omega,\mathcal{G}}$. Hence, from condition (1) and definition of tr, we conclude that:

  $$\mu_1 \in \llbracket (Q_1^*) \rrbracket_{\Omega,\mathcal{G}}, \quad \mu_2 \in \llbracket (Q_2^*) \rrbracket_{\Omega,\mathcal{G}}$$

  Thus, given that $\mu = \mu_1 \cup \mu_2$ by conditions (1), (2), (3) and definition of tr, we conclude that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$.
  Second, we show that $\llbracket Q^* \rrbracket_{\Omega,\mathcal{G}} \subseteq tr(\llbracket Q \rrbracket_I)$. Assume that $\mu \in \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Then there exist mappings μ_1, μ_2 such that: (1) $\mu = \mu_1 \cup \mu_2$, (2) $\mu_1 \in \llbracket (Q_1^*) \rrbracket_{\Omega,\mathcal{G}}$, and (3) $\mu_2 \in \llbracket (Q_2^*) \rrbracket_{\Omega,\mathcal{G}}$. By induction hypothesis, we have that $\mu_1 \in tr(\llbracket Q_1 \rrbracket_I)$ and $\mu_2 \in tr(\llbracket Q_2 \rrbracket_I)$. Thus, there exist tuples $t_1 \in \llbracket Q_1 \rrbracket_I, t_2 \llbracket Q_2 \rrbracket_I$ such that $\mu_1 = tr(t_1)$ and $\mu_2 = tr(t_2)$. From conditions (1), (2), (3) and definition of tr, we have that $t_1.A_i = t_2.A_i = \mu(A_i) \neq NULL$ for every $i \in \{1, ..., l\}$. Thus, given that $(att(Q_1) \cap att(Q_2)) = \{A_1, ..., A_l\}$, we have that $t \in \llbracket Q \rrbracket_I$, where $t : (att(Q_1) \cup att(Q_2)) \to (\mathbb{D} \cup \{NULL\})$ such that: (4) $t.A_i = t_1.A_i = t_2.A_i$ for every $i \in \{1, ..., l\}$, (5) $t.A = t_1.A$ for every $A \in (att(Q_1) \setminus att(Q_2))$, and (6) $t.A = t_2.A$ for every $A \in (att(Q_2) \setminus att(Q_1))$. Hence, we conclude that $\mu \in tr(\llbracket Q \rrbracket_I)$, given that $\mu = tr(t)$ by definition of t, definition of tr and conditions (1),(2) and (3).
- **Union:** Assume that $Q = (Q_1 \cup Q_2)$ and $Q^* = (Q_1^* UNION Q_2^*)$. Next, we prove that $tr(\llbracket Q \rrbracket_I) = \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$.
  First, we show that $tr(\llbracket Q \rrbracket_I) \subseteq \llbracket Q^* \rrbracket_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr(\llbracket Q \rrbracket_I)$. Then there exists a tuple $t \in \llbracket Q \rrbracket_I$ such that

$tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ or $t \in [\![Q_2]\!]_I$. Without loss of generality, assume that $t \in [\![Q_1]\!]_I$. Then we have that $tr(t) \in tr([\![Q_1]\!]_I)$ and, hence, $tr(t) \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ by induction hypothesis. Therefore, $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ since $tr(t) = \mu$, from which we conclude that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$.
Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ or $\mu \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$. Without loss of generality, assume that $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$. Then, by induction hypothesis, we have that $\mu \in tr([\![Q_1]\!]_I)$, and, hence, there exists a tuple $t \in [\![Q_1]\!]_I$ such that $tr(t) = \mu$. Therefore, we conclude that $t \in [\![(Q_1 \cup Q_2)]\!]_I$, from which we deduce that $\mu \in tr([\![Q]\!]_I)$.

- **Difference:** Assume that $Q = (Q_1 \setminus Q_2)$, and that $att(Q_1) = att(Q_2) = \{A_1, ..., A_l\}$. Then for every (not necessarily nonempty) set $\mathcal{X} = \{i_1, i_2, ..., i_p\}$ such that $1 \leq i_1 < i_2 < ... < i_p \leq l$. Define $R_{\mathcal{X}}$ as the following filter condition:

$n.A_{i_1} \neq$ NULL AND $n.A_{i_2} \neq$ NULL AND ...
AND $n.A_{i_p} \neq$ NULL AND
$n.A_{j_1} =$ NULL AND $n.A_{j_2} =$ NULL AND ...
AND $n.A_{j_q} =$ NULL

where $1 \leq j_1 < j_2 < ... < j_q \leq l$ and $\{j_1, j_2, ..., j_q\} = (\{1, ..., l\} \setminus \{i_1, i_2, ..., i_p\})$. That is, condition $R_{\mathcal{X}}$ indicates that every variables A_i with $i \in \mathcal{X}$ is not null, while every variable A_j with $j \in (\{1, ..., l\} \setminus \mathcal{X})$ is null. Moreover, for every $\mathcal{X} \neq \emptyset$ define the query $P_{\mathcal{X}}$ as follows ($R \in \mathbf{I}$):

$P_{\mathcal{X}'}$ = SELECT $n_1.A$
MATCH $(n_1 : R)$
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
AND $n_1.A$ IN $R_{\mathcal{X}}$

$P_{\mathcal{X}''}$ = SELECT $n_2.B$
MATCH $(n_2 : R)$
ON $\mathcal{G}$
OPTIONAL (Q_2^*)
AND $n_2.B$ IN $R_{\mathcal{X}}$

$P_{\mathcal{X}} = P_{\mathcal{X}'}$ MINUS $P_{\mathcal{X}''}$

Notice that there are $2^l - 1$ possible queries $P_{\mathcal{X}}$ with $\mathcal{X} \neq \emptyset$. Let $P_1, P_2, ..., P_{2^l-1}$ be an enumeration of these queries. Moreover, assuming that X, Y, Z are fresh variables, let $P_\emptyset$ be the following query:

$P_{\emptyset'}$ = SELECT $n_1.A$
MATCH $(n_1 : R)$
ON $\mathcal{G}$
OPTIONAL (Q_1^*)
AND $n_1.A$ IN $R_\emptyset$

$P_{\emptyset''}$ = $n_1 : R$
OPTIONAL (Q_2^*)
AND $n_1.A$ IN $R_\emptyset$

$P_{\emptyset'''}$ = SELECT $n_1.X, n_1.Y, n_1.Z$
MATCH $(n_1.R)$

$P_\emptyset = P_{\emptyset'}$ OPTIONAL ($P_{\emptyset''}$ AND EXISTS ($P_{\emptyset'''}$ WHERE $n_1.X =$ NULL))

Next, we show that $tr([\![Q]\!]_I) = [\![Q^*]\!]_{\Omega,\mathcal{G}}$. In this proof, we assume, by considering Lemma 2, that for every mapping μ such that $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ or $\mu \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$, it holds that $dom(\mu) = \{A_1, ..., A_l\}$.
First, we show that $tr([\![Q]\!]_I) \subseteq [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Assume that $\mu \in tr([\![Q]\!]_I)$. Then there exists a tuple $t \in [\![Q]\!]_I$ such that $tr(t) = \mu$. Thus, we have that $t \in [\![Q_1]\!]_I$ and $t \notin [\![Q_2]\!]_I$, from which we conclude by considering the induction hypothesis that $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \notin [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$. We consider two cases to show that this implies that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$.

- Assume that $dom(\mu) \neq \emptyset$, and let $\mathcal{X} = \{i \in \{1, ..., l\} | A_i \in dom(\mu)\}$. Given that $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$, we have that $dom(\mu) \subseteq \{A_1, ..., A_l\}$ and, hence, $\mathcal{X} \neq \emptyset$. Furthermore, we have that $\mu \models R_{\mathcal{X}}$ and, hence, $\mu \in [\![P_{\mathcal{X}'}]\!]_{\Omega,\mathcal{G}}$. From this and the fact that $\mu \notin [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$, we conclude that:
$$\mu \in [\![P_{\mathcal{X}}]\!]_{\Omega,\mathcal{G}} \text{ (Diff1)}$$
To see why this is the case, assume that (Diff1) does not hold. Then given that $\mu \in [\![P_{\mathcal{X}'}]\!]_{\Omega,\mathcal{G}}$, we conclude by the definition of the operator MINUS that there exists a mapping $\mu' \in [\![P_{\mathcal{X}''}]\!]_{\Omega,\mathcal{G}}$ such that $\mu \sim \mu'$ and $(dom(\mu) \cap dom(\mu')) \neq \emptyset$. Given that $\mu' \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$, we have that $dom(\mu') \subseteq \{A_1, ..., A_l\}$. Thus, given that $\mu' \models R_{\mathcal{X}}$ and $dom(\mu) \subseteq \{A_1, ..., A_l\}$, we conclude that $dom(\mu) = dom(\mu')$. Therefore, given that $\mu \sim \mu'$, we have that $\mu = \mu'$, from which we conclude that $\mu \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$, leading to a contradiction. From (Diff1) and definition of Q^*, we conclude that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$ since $P_{\mathcal{X}} = P_i$ for some $i \in \{1, ..., 2^l - 1\}$ (recall that $\mathcal{X} \neq \emptyset$).
- Assume that $dom(\mu) = \emptyset$. Then we have that $\mu \models R_{\emptyset}$ and, hence, $\mu \in [\![P_{\emptyset'}]\!]_{\Omega,\mathcal{G}}$. From this and the fact that $\mu \notin [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$, we conclude that:
$$\mu \in [\![P_{\emptyset}]\!]_{\Omega,\mathcal{G}} \text{ (Diff2)}$$
To see why this is the case, we assume that (Diff2) does not hold. Then given that $\mu \in [\![P_{\emptyset'}]\!]_{\Omega,\mathcal{G}}$ and $dom(\mu) = \emptyset$, we have that there exists a mapping $\mu' \in [\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}}$ such that $X \in dom(\mu')$. Thus, there exist mappings $\mu_1 \in [\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}}$ and $\mu_2 \in [\![P_{\emptyset'''}]\!]_{\Omega,\mathcal{G}}$ such that $\mu' = \mu_1 \cup \mu_2$. Given that $\mu_1 \in [\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}}$, we have that $\mu_1 \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$ and $\mu_1 \models R_{\emptyset}$. Thus, we have that $dom(\mu_1) \subseteq \{A_1, ..., A_l\}$, from which we conclude that $dom(\mu_1) = \emptyset$ (since $\mu_1 \models R_{\emptyset}$). Therefore, we have that $\mu = \mu_1$, which implies that $\mu \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$ and leads to a contradiction. From (Diff2) and definition of Q^*, we conclude that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$.

Second, we show that $[\![Q^*]\!]_{\Omega,\mathcal{G}} \subseteq tr([\![Q]\!]_I)$. Assume that $\mu \in [\![Q^*]\!]_{\Omega,\mathcal{G}}$. Then we consider two cases to prove that $\mu \in tr([\![Q]\!]_I)$.

- Assume that there exists $i \in \{1, ..., l\}$ such that $\mu \in [\![P_i]\!]_{\Omega,\mathcal{G}}$. Then there exists $\mathcal{X} \neq \emptyset$ such that $\mu \in [\![P_{\mathcal{X}}]\!]_{\Omega,\mathcal{G}}$. Thus, we have that $\mu \in [\![Q_1]\!]_{\Omega,\mathcal{G}}$ and $\mu \models R_{\mathcal{X}}$, from which we conclude that $\emptyset \subsetneq dom(\mu) \subseteq \{A_1, ..., A_l\}$. From this fact and the definition of the MINUS operator, we obtain that $\mu \notin [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$. Hence, by induction hypothesis, we conclude that $\mu \in tr([\![Q_1]\!]_I)$ and $\mu \notin tr([\![Q_2]\!]_I)$. That is, there exists a tuple t such that $tr(t) = \mu, t \in [\![Q_1]\!]_I$ and $t \notin [\![Q_2]\!]_I$, from which we conclude that $\mu \in tr([\![Q]\!]_I)$.
- Assume that (Diff2) holds. First, we show that $[\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}} = \emptyset$. For the sake of contradiction, assume that there exists a mapping $\mu' \in [\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}}$. Then given that $\mu' \in [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \models R_{\emptyset}$, we conclude that $dom(\mu') = \emptyset$. Given that $\mathcal{G}$ is a nonempty property graph and $dom(\mu') = \emptyset$, we conclude that there exists a mapping $\mu'' \in [\![P_{\emptyset''}$ AND EXISTS ($P_{\emptyset'''}$ WHERE $n_1.X =$ NULL)$]\!]_{\Omega,\mathcal{G}}$ such that $dom(\mu'') = \{X, Y, Z\}$. Thus, given that variables X, Y, Z are not mentioned in $P_{\emptyset'}$, we conclude that μ'' is compatible with every mapping in $[\![P_{\emptyset'}]\!]_{\Omega,\mathcal{G}}$. Thus, by definition of the OPTIONAL operator, we conclude that X belongs to the domain of every mapping in $[\![P_{\emptyset'}$ OPTIONAL ($P_{\emptyset''}$ AND EXISTS ($P_{\emptyset'''}$))$]\!]_{\Omega,\mathcal{G}}$ which implies that $[\![P_{\emptyset}]\!]_{\Omega,\mathcal{G}} = \emptyset$. But this leads to a contradiction as we assume that (Diff2) holds.
Given that (Diff2) holds and $[\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}} = \emptyset$, we conclude that $\mu \in [\![P_{\emptyset'}]\!]_{\Omega,\mathcal{G}}$ and $\mu \notin [\![P_{\emptyset''}]\!]_{\Omega,\mathcal{G}}$. Hence, we have that $\mu \in [\![Q_1^*]\!]_{\Omega,\mathcal{G}}$ and $\mu \notin [\![Q_2^*]\!]_{\Omega,\mathcal{G}}$ and therefore, we conclude by induction hypothesis, that $\mu \in tr([\![Q_1]\!]_I)$ and $\mu \notin tr([\![Q_2]\!]_I)$. That is, there exists a tuple t such that $tr(t) = \mu, t \in [\![Q_1]\!]_I$ and $t \notin [\![Q_2]\!]_I$, from which we conclude that $\mu \in tr([\![Q]\!]_I)$.

■

C. Proof of Proposition 1

Proof: Assume that we have a relational schema containing a relation with name *STUDENT* and attributes *SID*,*NAME*, and assume that the attribute *SID* is the primary key. Moreover, assume that this relation has two tuples, t_1 and t_2 such that $t_1.SID = 1, t_1.NAME = John, t_1.rId = 1$ and $t_2.SID = 1, t_2.NAME = Peter, t_2.rId = 2$. It is clear that the primary key is violated, therefore the database is inconsistent. If $\mathcal{DM}$ would be semantics preserving, then the resulting property graph would be inconsistent. However, the result of applying $\mathcal{DM}$, returns the following consistent property graph:

- $NODE("1", "STUDENT") \leftarrow REL("STUDENT"), VALUE("1", "SID", "1", "STUDENT")$
- $NODE("1", "STUDENT") \leftarrow REL("STUDENT"), VALUE("John", "NAME", "1", "STUDENT")$
- $NODE("2", "STUDENT") \leftarrow REL("STUDENT"), VALUE("1", "SID", "2", "STUDENT")$
- $NODE("2", "STUDENT") \leftarrow REL("STUDENT"), VALUE("Peter", "NAME", "2", "STUDENT")$
- $PROPERTY("1", "SID", "1") \leftarrow NODE("1", "STUDENT"), VALUE("1", "SID", "1", "STUDENT")$
- $PROPERTY("1", "NAME", "John") \leftarrow NODE("1", "STUDENT"), VALUE("John", "NAME", "1",$ $"STUDENT")$
- $PROPERTY("2", "SID", "1") \leftarrow NODE("2", "STUDENT"), VALUE("1", "SID", "2", "STUDENT")$
- $PROPERTY("2", "NAME", "Peter") \leftarrow NODE("2", "STUDENT"), VALUE("Peter", "NAME", "2",$ $"STUDENT")$

Therefore, $\mathcal{DM}$ is not semantics preserving.

■

D. *Proof of Proposition 2*

Proof: It is straightforward to see that a given relational schema $\mathbf{R}$, set Σ of (only) PKs over $\mathbf{R}$ and instance I of $\mathbf{R}$ such that $I \models \Sigma$, it holds that $\mathcal{DM}_{pk}(\mathbf{R}, \Sigma, I)$ is consistent. Likewise, if $I \not\models \Sigma$, then by definition of $\mathcal{DM}_{pk}$, the resulting property graph will have a duplicate node $NODE(id(t.rId, R), R)$, which would generate an inconsistency in the property graph structure. ■

E. *Proof of Theorem 3*

Proof: It is straightforward to see that a given relational schema $\mathbf{R}$, set Σ of (only) PKs over $\mathbf{R}$ and instance I of $\mathbf{R}$ such that $I \models \Sigma$, it holds that $\mathcal{DM}_{pk+fk}(\mathbf{R}, \Sigma, I)$ is consistent. Likewise, if $I \not\models \Sigma$, then by definition of $\mathcal{DM}_{pk+fk}$, the resulting property graph will have a duplicate node $NODE(id(t.rId, R), R)$, which would generate an inconsistency in the property graph structure.

■