Transforming Property Graphs

Angela Bonifati
Lyon 1 Univ., Liris CNRS & IUF
angela.bonifati@univ-lyon1.fr

Filip Murlak
Univ. of Warsaw
fmurlak@mimuw.edu.pl

Yann Ramusat
Lyon 1 Univ., Liris CNRS
yann.ramusat@liris.cnrs.fr

ABSTRACT

In this paper, we study a declarative framework for specifying transformations of property graphs. In order to express such transformations, we leverage queries formulated in the Graph Pattern Calculus (GPC), which is an abstraction of the common core of recent standard graph query languages, GQL and SQL/PGQ. In contrast to previous frameworks targeting graph topology only, we focus on the impact of data values on the transformations—which is crucial in addressing users' needs. In particular, we study the complexity of checking if the transformation rules do not specify conflicting values for properties, and we show this is closely related to the satisfiability problem for GPC. We prove that both problems are PSpace-complete.

We have implemented our framework in openCypher. We show the flexibility and usability of our framework by leveraging an existing data integration benchmark, adapting it to our needs. We also evaluate the incurred overhead of detecting potential inconsistencies at run-time, and the impact of several optimization tools in a Cypher-based graph database, by providing a comprehensive comparison of different implementation variants. The results of our experimental study show that our framework exhibits large practical benefits for transforming property graphs compared to ad-hoc transformation scripts.

PVLDB Reference Format:
Angela Bonifati, Filip Murlak, and Yann Ramusat. Transforming Property Graphs. PVLDB, 17(11): 2906 - 2918, 2024.
doi:10.14778/3681954.3681972

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yannramusat/TPG.

1 INTRODUCTION

Query languages for property graphs—those supported by existing systems, such as Neo4j's openCypher [22] or Oracle's PGQL [33], and those codified in international standards, such as GQL and SQL/PGQ [21]—define their semantics in terms of sets of tuples. This is inadequate for data interoperability tasks such as data migration or data integration, where outputs of some queries are to be fed directly to other queries. To support this kind of *composability*, queries should be able to output property graphs rather than sets of tuples. Such queries can be seen as *transformations*, turning an input property graph into an output property graph.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681972

Interoperability of graph data has received little attention so far, compared to the relational and XML data models [3]. Notable research in the area [6, 11] relies on the simplified graph data model that had been devised to provide the foundations for querying the topology of graphs with formalisms such as conjunctive regular path queries (CRPQs) [8] or regular queries [34]. As the simplified graph data model ignores the presence of properties (key-value pairs stored in nodes and edges), it is too far from the property graph models used in graph databases such as Neo4j or Tigergraph, and cannot be a foundation for practical solutions. These currently rely on opaque external libraries, such as Neo4j's APOC [28], or involve complex handcrafted queries [29], as illustrated below.

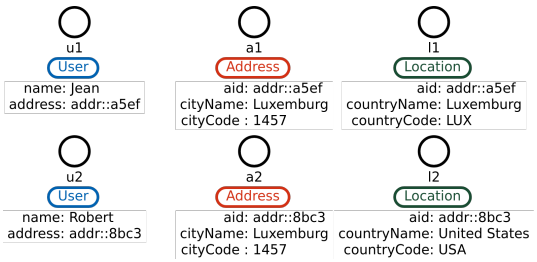
Example 1.1. Figure 1 illustrates a graph transformation scenario, in which a user has imported relational data into the popular Neo4j graph database and would like to reshape it into a semantically meaningful property graph instance, to facilitate navigational querying. The relational data consists of three tables,

$$\mathsf{User}(\underline{name}, \underline{address}), \ \mathsf{Address}(\underline{aid}, cityName, cityCode),$$
$$\mathsf{Location}(\underline{aid}, countryName, countryCode),$$

with primary keys consisting of the underlined attributes, and having two foreign keys: *aid* references *address* in User from both Address and Location.

Figure 1 (i) shows a rudimentary property graph obtained after importing the relational data, using a generic ingestion method, such as Cypher's `LOAD CSV` clause. In the resulting property graph, each node represents a single tuple of the relational instance, with the relation's name represented as the label, and the attributes stored in the node's properties. Note that there are no edges in this property graph: relationships between places, locations, and users are represented by way of foreign keys, just like in the original relational instance. Needless to say, this is not the best way to represent a relational instance as a property graph.

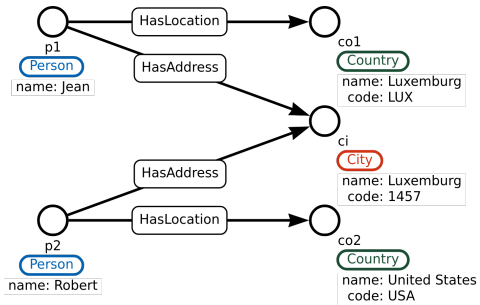
The user now wants to transform the instance in Figure 1 (i) into one that makes better use of the property graph data model by facilitating navigational operations in queries like "Which people live in the same city as Jean?". The user intends to create a node for each person, city, and country, and replace foreign key references with explicit relationships. Figure 1 (ii) shows an implementation of this transformation in openCypher that closely follows a graph refactoring solution described in Neo4j's GraphAcademy [29]. The reader will notice how difficult it is to relate the constructs of this query to the informal specification above. Even just making sense of the **`MERGE`** clauses interleaved with implicit grouping and list manipulations (**`UNWIND`** and `collect`) is a daunting task for an unacquainted user. But the query leverages other advanced idioms too. For instance, in Line 5, the script creates as many nodes of type `Person` as there are rows output by the previous **`WITH`** clause: one for each u, due to implicit grouping. In line 9, the script generates one `City` node for each *distinct* value found in property `cityName` across all



(i) Input property graph G containing ingested relational data.

```
1  MATCH (u:User)
2  MATCH (a:Address) WHERE a.aid = u.address
3  MATCH (l:Location) WHERE l.aid = u.address
4  WITH u, collect(a) AS Addresses, collect(l) AS Locations
5  CREATE (p:Person)
6  SET p.name = u.name
7  WITH p, Addresses, Locations
8  UNWIND Addresses AS a
9  MERGE (ci:City {name: a.cityName})
10 SET ci.code = a.cityCode
11 MERGE (p)-[:HasAddress]->(ci)
12 WITH p, Locations
13 UNWIND Locations AS l
14 MERGE (co:Country {name: l.countryName})
15 SET co.code = l.countryCode
16 MERGE (p)-[:HasLocation]->(co)
```

(ii) Ad-hoc transformation script in openCypher.



(iii) Resulting output property graph H.

Figure 1: Ad-hoc transformation of raw ingested data.

a's; this is because the property `name` is specified as `a.cityName` in the **MERGE** clause. Similarly, in line 14, a single `Country` node is created for each *distinct* value found in property `countryName`.

Figure 1 (iii) shows the output property graph obtained by running the script on the input property graph from Figure 1 (i). It reveals that the ad-hoc transformation fails to account for the fact that cities are weak entities that cannot be identified by their name alone, and conflates Luxemburg in Europe with Luxemburg in the US. Detecting such errors is hard because openCypher lacks a transparent mechanism for specifying identities of created elements. ◀

As we have seen, ad-hoc transformation scripts are error-prone and hard to interpret and analyze. Moving away from handcrafted implementations to declarative specifications has long been recognized as pivotal for solving data programmability problems [10]. The aim of this work is to lay the theoretical foundations for the declarative specification of property graph transformations, and facilitate practical solutions for turning such specifications into executable scripts in modern property graph query languages. Constraint-based, fully declarative formalisms, such as schema mappings for relational [9, 16, 17] and graph [6, 23] data, allow multiple target solutions, leading to ambiguous transformations [12, 14]. For property graphs, this makes the schema mapping problem undecidable even under strong restrictions [23]. We avoid this problem by focusing on transformations that return a unique, well-defined output instance for each input instance, thus facilitating direct execution of the transformation.

We propose a rule-based formalism that allows the user to describe the output property graph based on the input property graph, by specifying not only labels, properties, and relationships between output elements, but also their identities. The formalism builds upon the *Graph Pattern Calculus* (GPC) [20], which is an abstraction of the common graph pattern matching features of GQL and SQL/PGQ [15]. GPC is adequate in terms of expressive power: it has ample facilities for querying properties and even on property-less graphs it goes well beyond classical formalisms such as RPQs and CRPQs. It is suitable for theoretical investigation owing to its concise syntax and rigorous semantics. It should also keep our proposal future-proof by ensuring out-of-the-box compatibility with the expected implementations of these standards. Until then, we can rely on the already implemented graph query languages, such as Neo4j's openCypher [22] or Oracle's PGQL [33], which were a strong inspiration for GQL. Indeed, the actual query language used in the rules can be seen as a parameter of the framework.

In contrast to the purely topological formalism of [11], specifications of property-aware transformations can easily become *inconsistent*, when they attempt to specify two different values for the same property of a given element. Detecting such conflicts naturally comes to the foreground of static analysis. As we show, this problem is tightly connected to the *satisfiability* problem for GPC+ (GPC extended with union and projection, also introduced in [20]), which is to decide if there is a property graph satisfying a given GPC+ query. Exploiting this connection, we establish tight complexity bounds for both these problems, showing that they are PSpace-complete. To the best of our knowledge, this is the first static-analysis result on GPC. Given that query satisfiability is the work horse of static analysis throughout database theory, we believe that with the adoption of the GQL standard our result will find other uses. An immediate consequence for property graph transformations is that consistency cannot be checked statically due to the prohibitive cost, and conflicts must be handled dynamically, during the execution of the transformation.

In order to prove that our formalism can serve as a foundation for practical data interoperability solutions, we provide a proof-of-concept implementation. As no existing query engine supports GQL yet, we rely on the Neo4j's open-source implementation of openCypher [22, 25], which offers most of the functionalities of GQL described in [21]. We study the case when the rules are provided by the users and describe a generic, easily automated method of translating these rules into executable openCypher scripts, and apply it manually to selected realistic property graph transformations derived from real-world data integration scenarios of the iBench benchmark suite [4]. We perform a comprehensive experimental study gauging the efficiency of conflict detection and the

effect of rule order and various optimizations on several implementation variants. We confirm that our implementation performs well in all scenarios and scales to large input data. We also demonstrate that our framework can be successfully applied in a concrete data integration scenario on real-world data [19], and report the results of a small-scale user study confirming that our framework enhances readability and usability of transformations.

In summary, our main contributions are the following.

- We propose a comprehensive declarative formalism for specifying transformations of property graphs, compatible with SQL/PGQ and the upcoming GQL standard.
- We identify consistency as a key static-analysis problem, and show that it is interreducible with satisfiability of GPC+ queries and that both are PSpace-complete.
- We provide a proof-of-concept implementation of our formalism in openCypher, and apply it to realistic scenarios of graph-shaped data transformations.
- We show experimentally that our solution scales to large input data, handles on-the-fly conflict detection with low overhead, and enhances readability and usability, without sacrificing preformance.

The rest of paper is organized as follows. In Section 2, we recall the property graph data model along with GPC. In Section 3, we give syntax and semantics of our graph transformation formalism. In Section 4, we discuss the consistency in relation with satisfiability of GPC+ queries, and establish the complexity bounds. In Section 5, we describe our-proof-of-concept implementation. In Section 6, we present both the experiments and the user study. In Section 7 and in Section 8, we discuss the related work and conclude the paper.

2 PRELIMINARIES

We briefly introduce the basic concepts of the property graph data model and the *Graph Pattern Calculus* (GPC) that we use in this paper. We mostly follow the definitions from [20].

2.1 Data model

Conforming to the formal specification originating from [20], a *property graph* G is a tuple $\langle N, E, \lambda, \mathrm{src}, \mathrm{tgt}, \delta\rangle$ where $\mathcal{O}$, $\mathcal{L}$, $\mathcal{K}$ and Const are disjoint countable sets of object identifiers (ids), labels, keys (also called property names) and constants (data values), and

- $N \subset \mathcal{O}$ is the finite set of node ids in G;
- $E \subset \mathcal{O}$ is the finite set of edge ids;
- N and E are disjoint;
- $\lambda : N \cup E \to 2^{\mathcal{L}}$ is a labeling function that associates to every id a (possibly empty) finite set of labels;
- $\mathrm{src}, \mathrm{tgt} : E \to N$ define the source and target of each edge;
- $\delta : (N \cup E) \times \mathcal{K} \to \mathrm{Const}$ is a finite-domain partial function that associates a constant with an id and a key from $\mathcal{K}$.

The node ids and edge ids will be respectively called the *nodes* and *edges* of the property graph.

That is to say, a property graph is a *multigraph* in the sense that two vertices may be connected by more than one edge, even with these edges having the same label(s), and that loops are permitted. All the *elements* of the database (the nodes and the edges) store a finite set of property-value pairs, represented by δ.

A property graph is presented in Figure 1 (iii). It contains information about peoples' location such as the City and the Country they live in. We see that it contains one node with label City and two nodes with label Country; two edges with label HasLocation and two edges with label HasAddress; all nodes have property *name*; and all nodes with label City or Country have an additional property *code*. Annotations p1, p2, ..., co2 are node identifiers; edge identifiers are not shown.

2.2 Graph Pattern Calculus

In the following, we introduce GPC by means of examples. The reader can refer to [20] for more details on GPC, and to [21] for insight on how it will actually be used in GQL.

In Example 1.1, the user can retrieve from the property graph in Figure 1 (iii) "all people who live in the same city as a person named $\$name$" using the following GPC query:

$$\underset{\langle name=\$name\rangle}{(:\mathsf{Person})} \xrightarrow{:\mathsf{HasAddress}} (:\mathsf{City}) \xleftarrow{:\mathsf{HasAddress}} (y:\mathsf{Person})$$

This is an example of a *path pattern*, which is essentially a regular path query [8] augmented with *conditioning*: the filter $\langle name = \$name\rangle$ checks that the value of the property *name* is indeed the one sought. Given a property graph, this pattern returns the nodes that can be matched to y.

One can also use *graph patterns* in GPC (also called patterns or queries in this paper), which are conjunctions of path patterns. For example, the following query retrieves pairs of people living in the same city, such that one person knows, possibly indirectly, the other one:

$$(x:\mathsf{Person}) \xrightarrow{:\mathsf{HasAddress}} (:\mathsf{City}) \xleftarrow{:\mathsf{HasAddress}} (y:\mathsf{Person}),$$

$$(x) \xrightarrow{:\mathsf{Knows}}{}^{1..\infty} (y)\,.$$

Such graph patterns generalize conjunctive two-way regular path queries [8] to property graphs.

In GPC, each path pattern occurring in a graph pattern must be qualified with a *restrictor* among the set of simple, trail (used by default if none is given) and shortest. The restrictor's purpose is to ensure a finite result set: simple prevents repetition of nodes along a path; trail prevents repetition of edges; and shortest selects only the paths of minimal length among all the paths between two nodes.

For the ease of exposition, we simplify the semantics of GPC. We assume that a pattern only returns a set of bindings (in [20], a tuple of witnessing paths is also returned with each binding). In GPC, variables used in the scope of a repetition operator, such as $1..\infty$, are called *group variables* and are bound to lists of nodes or edges. The remaining variables are called *singleton variables* and are bound to single nodes or edges. For the purpose of our transformation formalism we restrict the output of queries to singleton variables.

For a GPC pattern P, a tuple $\bar{x}$ of singleton variables in P, and a property graph G, we write $[\![P]\!]^{\bar{x}}_G$ for the set of bindings of $\bar{x}$ returned by P on G. For instance, if P is the first query above and G is the property graph depicted in Figure 1 (iii), we have $[\![P]\!]^{y}_G = \{(y \mapsto \mathsf{p2})\}$ when $\$name$ is *"Jean"*, $[\![P]\!]^{y}_G = \{(y \mapsto \mathsf{p1})\}$

when $\$name$ is *"Robert"*, and $[\![P]\!]_G^y = \emptyset$ for any other name. (Note that the trail restrictor has been used by default.)

3 PROPERTY GRAPH TRANSFORMATIONS

In this section, we present our declarative formalism for specifying property graph transformations. An example is given in Figure 2. The specification consists of two rules. Each rule collects data from the input graph with a GPC pattern on the left of $\Longrightarrow$, and specifies elements of the input graph using the expression on the right. This expression resembles a GPC pattern, but it has specifications of the element's property values instead of filters and specifications of element identifiers instead of variables to be matched (new variables will reappear on the right-hand side, in a slightly different role). In what follows we discuss how new identifiers are generated using *Skolem functions* (Section 3.1) and how identifiers, labels, and properties of output elements are specified using *content constructors* (Section 3.2). Then, we describe the general form of rules (Section 3.3) and explain their semantics in terms of a procedure that generates an output property graph given an input property graph (Section 3.4). We shall also see if the transformation in Figure 2 fixes the issues discussed in Example 1.1.

3.1 Generating output identifiers

Throughout the paper we assume that all identifiers in input property graphs come from a countable set $\mathcal{S} \subset \mathcal{O}$ of *input identifiers*, and ensure that all identifiers in output property graphs come from a countable set $\mathcal{T} \subset \mathcal{O} \setminus \mathcal{S}$ of *output identifiers*. Following [11], to generate identifiers in the output graph, we use Skolem functions. Specifically, we use a fixed injective Skolem function

$$f : \bigcup_{k \in \mathbb{N}} (\mathcal{O} \cup \text{Const} \cup \mathcal{L})^k \to \mathcal{T}.$$

In the context of relational schema mappings and data exchange, Skolem functions are used for *value invention* [5], e.g., to generate artificial primary keys of new tuples in a way that makes is possible to refer to them in foreign keys. The way we use Skolem functions is similar, but not the same, because element identifiers are not data values. Rather, they are the property-graph analogue of object identifiers from the object-oriented data model [1, 26]. Most of the time they are invisible to the user, and are not expected to carry any information beyond the identity of the element. Thus, the specific choice of function f is truly irrelevant, as long as f is injective.

Example 3.1. In the rules in Figure 2 the Skolem function is kept implicit, but its arguments are explicitly listed. For example, in the subexpression $((u) : \text{Person})$, on the right-hand side of both rules, (u) indicates that the identifier of the ouptut node is $f(u)$ where u is (the identifier of) a node selected from the input property graph by the left-hand side GPC pattern, such as u1. Because the same nodes u are selected in both rules, the subexpressions $((u) : \text{Person})$ in both rules will be referring to the same output nodes. Further, $(\ell.countryName)$ specifies the node identifier as $f(\ell.countryName)$, where $\ell.countryName$ name refers to the value of the property $countryName$ in a node ℓ selected from the input graph, such as *"United States"*, and similarly for $(a.cityName)$. If $\ell.countryName = a.cityName$ for some ℓ and a, which can happen in our example, $(\ell.countryName)$ and $(a.cityName)$ will indicate the same output node. ◂

3.2 Content constructors

A property graph transformation must be able to specify not only the identifiers of output elements, but also their labels and properties. For this purpose, we use content constructors. A *content constructor* is an expression of the form:

$$\begin{aligned} C(\bar{x}) \coloneqq \{ & \\ \text{Id: } & (a_1, \ldots, a_k) \\ \text{Labels: } & L \\ \text{Properties: } & \langle k_1 = v_1, \ldots, k_n = v_n \rangle \ \} \end{aligned}$$

where $\bar{x}$ is a tuple of variables, L is a finite set of labels; each k_i is a property name from $\mathcal{K}$; each v_i is either a data value $c \in \text{Const}$, or an expression of the form $x.a$ for $x \in \bar{x}$ and $a \in \mathcal{K}$; and each a_i is either a constant $c \in \text{Const}$, or a label $\ell \in \mathcal{L}$, or an expression of the form $x.a$ or x for $x \in \bar{x}$ and $a \in \mathcal{K}$. The field Id specifies the identity of the node by listing the arguments to be fed to the Skolem function. The fields Labels and Properties specify labels and properties present in an element. Importantly, they do not forbid additional labels and properties, which will allow the user to split the description of an element across multiple rules, if the user so desires. We write C.Id for the content of the Id field of C, and similarly for other fields. When $\bar{x}$ is clear from the context, we simply write C instead of $C(\bar{x})$.

Example 3.2. In the first rule in Figure 2, new Country nodes are described using the following content constructor:

$$\begin{aligned} C_{\text{t}}(a, u, \ell) \coloneqq \{ & \\ \text{Id: } & (\ell.countryName) \\ \text{Labels: } & \{\text{Country}\} \\ \text{Properties: } & \langle name = \ell.countryName,\ code = \ell.countryCode \rangle \ \}. \end{aligned}$$

It specifies the identities and the values of properties *name* and *code* of new Country nodes in terms of the values of properties *countryName* and *countryCode* retrieved from elements to which variable ℓ is bound in the input graph. Rather then using the abstract syntax introduced above, the rule in Figure 2 presents C_{t} in GPC-like syntax [20] as

$$\underset{\langle name=\ell.countryName,\ code=\ell.countryCode \rangle}{((\ell.countryName) : \text{Country})} .$$

3.3 Transformations

We describe transformations in terms of property graph transformation rules. Each rule brings together the data retrieved from the input property graph by a GPC pattern and a description of output elements expressed with content constructors.

We recall that the semantics of GPC is defined such that a query returns *tuples*. Each tuple represents a *binding* of singleton variables in that query to elements of the property graph.

We have two kinds of *property graph transformation rules*: node rules and edge rules. A *node rule* is an expression of the form:

$$P(\bar{x}) \implies (C(\bar{x}))$$

where $P(\bar{x})$ is a GPC query with singleton variables $\bar{x}$ and $C(\bar{x})$ is a content constructor. An *edge rule* is an expression of the form:

$$P(\bar{x}) \implies (C_s(\bar{x})) \xrightarrow{C(\bar{x})} (C_t(\bar{x}))$$

where $P(\bar{x})$ is a GPC query with singleton variables $\bar{x}$ and $C_s(\bar{x})$, $C(\bar{x})$ and $C_t(\bar{x})$ are content constructors. Finally, a *property graph transformation* is a finite set of property graph transformation rules.

Example 3.3. The first edge rule in Figure 2 is built from the content constructor C_t as defined in Example 3.2, and of the following two content constructors C_s and C:

$C_s(a,u,\ell) :=$ {	$C(a,u,\ell) :=$ {
Id: (u)	Id: $()$
Labels: {Person}	Labels: {HasLocation}
Properties: $\langle name = u.name \rangle$ },	Properties: $\langle\rangle$ }. ◀

The above definition allows specifying multiple labels with a single constructor as well as specifying the labels of a single element using multiple rules. This feature, illustrated in the following example, is crucial. Without it, in the presence of type hierarchies, one would need negation in the query language to avoid duplicating output elements. In our setting, GPC does not permit negating patterns and it is unlikely for the complexity upper bounds in Section 4.1 to hold when this form of negation is added.

Example 3.4. As discussed in Example 3.1, if for some nodes ℓ and a selected by the GPC patterns in the rules of Figure 2, $\ell.countryName$ and $a.cityName$ are equal, then the C_t constructors in both rules refer to the same output node. For instance for, ℓ = l1 and a = a1 in the input graph in Figure 1 (i), both rules refer to the node $f(\text{“}Luxemburg\text{”})$ in the output graph in Figure 3. In consequence, this node has two labels, City and Country. This, quite likely, is not what the user actually wants. We will later see how to fix it by adjusting the rules. ◀

Property graphs are *multigraphs* and our rules allow specifying multiple edges with the same endpoints by using different arguments for the Skolem function. We will see an example in Section 5.

We refer to the right-hand side expressions in node (resp. edge) rules as node (resp. edge) constructors. We also allow rules of a more general form, illustrated in Figure 4, where a comma-separated list of node and edge constructors can be used on the right-hand side. We also support aliasing, with scope limited to a single rule. For instance, in the rule in Figure 4, we introduce alias $x = (u)$ in the first edge constructor, and use it in the second edge contructor. Both these extensions are syntactic sugar. To eliminate aliases, we simply substitute them with their definitions: in the example, we replace x in the second edge constructor with (u). Then, we split the rules: for each node or edge constructor on the right-hand side, we create a separate rule with the same GPC pattern on the left.

3.4 Semantics

In this section, we describe operationally in Algorithm 1 how a transformation given as a set of node and edge rules turns an input property graph into an output property graph. In Section 5, we will see how to implement this efficiently in an existing graph database.

Given a GPC query $P(\bar{x})$, a content constructor $C(\bar{x})$ and a binding $\bar{o}$ for $P(\bar{x})$ over an input property graph G, we define $C.\mathrm{Id}(\bar{o})$ by replacing in $C.\mathrm{Id}$ each x_j with o_j and each $x_j.a$ with $\delta_G(o_j, a)$. Similarly, we define $C.\mathrm{Properties}(\bar{o})$ by replacing in $C.\mathrm{Properties}$ each $x_j.a$ with $\delta_G(o_j, a)$.

Algorithm 1 Semantics of a set of transformation rules.

Input: A property graph G and a set of transformation rules T.
Output: An *output of the transformation T over G*, a property graph $T(G) = \langle N, E, \lambda, \mathrm{src}, \mathrm{tgt}, \delta \rangle$.

1: initialize $T(G)$ to the empty property graph
2: **for** each edge rule $P(\bar{x}) \implies (C_s(\bar{x})) \xrightarrow{C(\bar{x})} (C_t(\bar{x})) \in T$ **do**
3: add rules $P(\bar{x}) \implies (C_s(\bar{x}))$ and $P(\bar{x}) \implies (C_t(\bar{x}))$ to T
4: **for** each node rule $P(\bar{x}) \implies (C(\bar{x})) \in T$ **do**
5: **for** each binding $\bar{o} \in [\![P]\!]^{\bar{x}}_G$ **do**
6: $N \leftarrow N \cup \{o := f(C.\mathrm{Id}(\bar{o}))\}$
7: $\lambda(o) \leftarrow \lambda(o) \cup C.\mathrm{Labels}$
8: set $\delta(o,k)$ to c if $C.\mathrm{Properties}(\bar{o})$ sets property k to c
9: **for** each edge rule $P(\bar{x}) \implies (C_s(\bar{x})) \xrightarrow{C(\bar{x})} (C_t(\bar{x})) \in T$ **do**
10: **for** each binding $\bar{o} \in [\![P]\!]^{\bar{x}}_G$ **do**
11: $o_s \leftarrow f(C_s.\mathrm{Id}(\bar{o}))$; $o_t \leftarrow f(C_t.\mathrm{Id}(\bar{o}))$
12: $E \leftarrow E \cup \{o := f(o_s, C.\mathrm{Id}(\bar{o}), o_t)\}$
13: $\mathrm{src}(o) \leftarrow o_s$; $\mathrm{tgt}(o) \leftarrow o_t$
14: $\lambda(o) \leftarrow \lambda(o) \cup C.\mathrm{Labels}$
15: set $\delta(o,k)$ to c if $C.\mathrm{Properties}(\bar{o})$ sets property k to c

Example 3.5. We describe, step by step, the operations carried out by Algorithm 1 on the input consisting of the property graph G from Figure 1 (i) and the transformation T_1 which contains only the first of the two rules in Figure 2.

First, the GPC query

$$P(u,a,\ell) := (u : \mathsf{User}), (a : \mathsf{Address}), (\ell : \mathsf{Location}) \quad \langle u.address{=}a.aid,\, u.address{=}\ell.aid \rangle$$

is executed on G (only once in the entire process) and outputs the set of bindings $[\![P]\!]^{u,a,\ell}_G = \{(u \mapsto \mathsf{u1}, a \mapsto \mathsf{a1}, \ell \mapsto \mathsf{l1}), (u \mapsto \mathsf{u2}, a \mapsto \mathsf{a2}, \ell \mapsto \mathsf{l2})\}$.

In Line 3, the single edge rule of T_1 is split into node rules $P(\bar{x}) \implies (C_s(\bar{x}))$ and $P(\bar{x}) \implies (C_t(\bar{x}))$, where C_s and C_t have been defined in Example 3.3 and 3.2, respectively. These two node rules are added to T_1, which initially contains no node rules.

Suppose that the node rule $P(\bar{x}) \implies (C_s(\bar{x}))$ is considered first in the loop in Line 4. Two output nodes are created with respective identifiers $f(u1)$ and $f(u2)$ (Line 6), one for each binding. Initially, they have no labels, $\lambda(f(u1)) = \lambda(f(u2)) = \emptyset$, and no properties. Then both get label Person (Line 7) and their property *name* is set to *“Jean”* and *“Robert”*, respectively (Line 8).

Next, the algorithm moves to the node rule $P(\bar{x}) \implies (C_t(\bar{x}))$. Two nodes are created in the output with respective identifiers $f(\text{“}Luxemburg\text{”})$ and $f(\text{“}United\ States\text{”})$ (Line 6), one for each binding; they both get label Country (Line 7); and their properties *name* and *code* are filled in (Line 8).

Finally, the algorithm steps through the only edge rule in T_1. For the first binding, the nodes corresponding to the endpoints of the edge that has to be created, namely $o_s := f(u1)$

$$(u : \text{User}), (a : \text{Address}), (\ell : \text{Location}) \underset{\langle u.address=a.aid,\, u.address=\ell.aid\rangle}{\Longrightarrow} ((u) : \text{Person}) \underset{\langle name=u.name\rangle}{\xrightarrow{:\text{HasLocation}}} ((\ell.countryName) : \text{Country}) \underset{\langle name=\ell.countryName,\, code=\ell.countryCode\rangle}{} \quad (1)$$

$$(u : \text{User}), (a : \text{Address}), (\ell : \text{Location}) \underset{\langle u.address=a.aid,\, u.address=\ell.aid\rangle}{\Longrightarrow} ((u) : \text{Person}) \underset{\langle name=u.name\rangle}{\xrightarrow{:\text{HasAddress}}} ((a.cityName) : \text{City}) \underset{\langle name=a.cityName,\, code=a.cityCode\rangle}{} \quad (2)$$

Figure 2: Transformation T given as a set of rules.

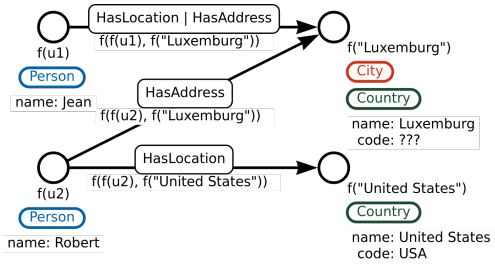


Figure 3: Output property graph $T(G)$.

and $o_t := f(\text{“}Luxemburg\text{”})$ are retrieved (Line 11). They correspond to the nodes that were created, from this binding, by the node rules that were added to T_1 in Line 3. An edge with id $f(f(u1), f(\text{“}Luxemburg\text{”}))$ is created (Line 12); its source and target are set to $f(u1)$ and $f(\text{“}Luxemburg\text{”})$, respectively (Line 13); it gets label HasLocation (Line 14); and no property is filled (Line 15). For the second binding, an edge is created by the same process between the nodes $f(u2)$ and $f(\text{“}United\ States\text{”})$. ◀

The role of Algorithm 1 is to give semantics to a set of transformation rules: it explains how the outputs of the multiple rules are consolidated into a single output property graph. The following result shows that our transformations are indeed graph-to-graph transformations, offering a way to meet the expected requirements of future versions of standard graph query languages [21].

Proposition 3.6. *Given an input property graph G and a property graph transformation T, Algorithm 1 always returns a valid instance of the property graph data model.*

Although Algorithm 1 always returns a valid property graph (Proposition 3.6), property values may depend on the order in which the rules and bindings are considered in Lines 4–5 and 9–10. Hence, the result of the transformation may be ill-defined on some inputs. We investigate this further in the next section.

4 DETECTING CONFLICTS

As one would expect from any expressive property graph transformation language, our formalism supports manipulating properties of output nodes and edges. Compared to purely structural mechanisms, such as [11], this poses additional challenges.

Example 4.1. Let us continue Example 3.5 by now considering the two-rule transformation T presented in Figure 2. The second rule gets split into two nodes rules, one of which is

$$P(\bar{x}) \Longrightarrow \underset{\langle name=a.cityName,\, code=a.cityCode\rangle}{((a.cityName) : \text{City})} .$$

Suppose that this node rule is processed in Line 4 after the two node rules discussed in Example 3.5. The algorithm attempts twice to create a node with identifier $f(\text{“}Luxemburg\text{”})$ (Line 6), once for each binding. However, a node with identifier $f(\text{“}Luxemburg\text{”})$ has been already created by the second node rule in Example 3.5. In consequence, the label City is added to this node (Line 7) and its properties *name* and *code* are set to *Luxemburg* and 1457, respectively (Line 8), overriding previous values *Luxemburg* and *LUX*. This means that one of the two values of property *code* is lost and it depends on the processing order of rules which one it is. Indeed, the mapping now conflates not only two cities called Luxemburg, as in Example 1.1, but also the country Luxemburg. This time, however, the error is easy to spot: looking at the rules we see immediately that the identity of the output nodes depends exclusively on the name of the city/country, which means that all cities and countries with the same name are conflated. We can fix the transformation easily by including information about the corresponding country in the identity of each City node, for instance by replacing $(a.cityName)$ with $(a.cityName, \ell.countryName)$ in rule (2) in Figure 2. ◀

Detecting the modelling error in the rules in Figure 2 requires human insight (basic understanding of geography) but we hope to make it easier by insisting on explicit identity specification in transformations. On the other hand, setting an output property to conflicting values is something one can try to capture abstractly and detect automatically. This is what we do next. In the reminder of this section we focus on detecting conflicts statically, by analysing a set of transformation rules to check if it can exhibit this pathological behavior on some input. We come back to handling conflicts dynamically in Section 5 and Section 6. Due to the limited space, most proofs are moved to the extended version of our paper [13].

4.1 Consistency

By a *conflict* we mean a situation when Algorithm 1 resets a previously set property to a different value, as illustrated in Example 4.1. A transformation T is *consistent* if for every input property graph G, no execution of Algorithm 1 results in a conflict. Note that even a transformation consisting of a single rule can be inconsistent, because different bindings for the same rule can cause a conflict.

We study the following fundamental static analysis problem, in the setting where there is no source schema constraining the set of possible input property graphs.

Consistency. Given a transformation T, check if T is consistent.

As we show next, consistency of transformations is deeply related to satisfiability of GPC patterns. A GPC pattern is *satisfiable* if it returns a non-empty set of answers on some property graph. Towards the goal of establishing complexity lower bounds for the consistency problem, we provide a polynomial-time reduction from the satisfiability problem for GPC.

Satisfiability. Given a GPC pattern P, check if P is satisfiable.

Lemma 4.2. *The satisfiability problem for GPC is PTime-reducible to the transformation consistency problem.*

Proof. For a GPC pattern P, let T_P be the transformation consisting of the following two rules

$$P() \implies \underset{\langle k=5 \rangle}{((c) : \ell)} \quad \text{and} \quad P() \implies \underset{\langle k=7 \rangle}{((c) : \ell)}$$

for some fixed label ℓ, constant c, and property name k. These rules are not conflicting with themselves, because their node constructors do not depend on the binding. However, they are conflicting with each other on a graph G if P returns at least one answer on G. Hence, T_P is consistent iff P is not satisfiable. □

For the converse of Lemma 4.2 to hold we need to move to GPC+, a simple extension of GPC with projection and union [20].

Lemma 4.3. *The transformation consistency problem is PTime-reducible to the satisfiability problem for GPC+.*

Proof sketch. For a pair of rules R and S, and an attribute a, we can write a Boolean GPC+ query $Q_{R,S,a}()$ that detects if some matches of R and S lead to different values for attribute a in the same element of the output graph. Because there are polynomially many such triples, we can take the union of all such queries to obtain the final GPC+ query to be checked for satisfiability. □

We now turn to study the complexity of the satisfiability problem for GPC and GPC+. The two lemmas above will allow us to draw conclusions for the consistency problem in Section 4.3.

4.2 The complexity of satisfiability

In Theorem 4.5, we establish that checking if a GPC+ query is satisfiable is a PSpace-complete problem (modulo certain assumptions on the use of restrictors). We believe that this result is interesting in its own right, beyond the application to transformation consistency we consider this paper. Indeed, deciding whether a query expressed in a given query language is satisfiable is a fundamental problem in database theory. Very little is known to this date about GPC from a theoretical viewpoint, and our work is one of the first to tackle a key static analysis task related to this query language.

Lemma 4.4. *The satisfiability problem for GPC is PSpace-hard.*

Proof sketch. We show how to reduce the membership problem for an arbitrary PSpace language to the satisfiability of a GPC query. Let L be a language in PSpace and M a deterministic polynomial-space Turing machine that recognizes L in space $c \cdot p(n)$ for a fixed constant c and polynomial p. In the following, n denotes the length of the word w which is an input to M.

We construct the following GPC pattern P:

$$P() \coloneqq \rho\ (x)_{\langle \theta_1 \rangle} \left(\left[(u) \to (v) \right]_{\langle \theta_2 \rangle} \right)^{1..\infty} (y)_{\langle \theta_3 \rangle}$$

The intuition is the following. We can represent a configuration of M in a single node, using a polynomial number of properties. The pattern $(x)_{\langle \theta_1 \rangle}$ is responsible for encoding the initial configuration of M over the input word w. The pattern $\left[(u) \to (v) \right]_{\langle \theta_2 \rangle}$ ensures that there exists a valid transition of M between the configurations represented by nodes u and v. Finally, $(y)_{\langle \theta_3 \rangle}$ specifies that node y represents an accepting configuration.

We can use techniques similar to the proof of the Cook-Levin Theorem [24] to construct in time polynomial in n the formulæ θ_1, θ_2, and θ_3. The size of P is then clearly polynomial in n. This reduction works with any $\rho \in \{\text{shortest}, \text{simple}, \text{trail}\}$. □

For completeness, we also provide the matching upper bound (under some assumptions) and obtain Theorem 4.5 as a result. The details of the upper-bound proof are highly technical and the claim depends heavily on the design choices made for GPC.

Theorem 4.5. *The satisfiability problem for GPC+ queries using only the* simple *and* trail *restrictors is PSpace-complete.*

We prove the upper-bound of Theorem 4.5 by inductively constructing an equality type over all variables in the query. This nondeterministic procedure uses only a polynomial amount of space by avoiding storing the full match of the pattern. Unfortunately, this does not extend to queries using the shortest restrictor: they seem to require storing the full match. We leave open the question of pinpointing the exact complexity of satisfiability for such queries.

Given the high complexity lower bounds, one might wonder whether there are useful subclasses of GPC with tractable satisfiability. In Lemma 4.6 below, we show that even under strong limitations, satisfiability is still intractable.

Lemma 4.6. *The satisfiability problem is NP-hard even for single-node GPC patterns.*

4.3 Back to consistency

From Theorem 4.5, Lemma 4.2, and Lemma 4.3, we obtain the following fundamental result.

Corollary 4.7. *The consistency problem is PSpace-complete for transformations using only* simple *and* trail *restrictors.*

In fact, the PSpace lower bound holds already for transformations using only two rules and any single restrictor. From Lemma 4.6 and Lemma 4.2 it follows that the problem remains intractable even for transformations using very restricted GPC queries.

In the light of these high complexity lower bounds, it is unlikely that conflict detection can be handled statically in practice. This means that conflicts have to be handled dynamically, when the transformation is executed. In Section 5 we discuss how this can be implemented in practice and in Section 6 we show experimentally that the incurred overhead is affordable.

5 TRANSLATION TO CYPHER

Algorithm 1 can be seen as an abstraction of a transformation engine: it takes a transformation and an input property graph, and produces an output property graph. In this section we show how to compile a transformation to an openCypher script that can be directly executed in any openCypher engine. This is similar in spirit to executable SQL scripts for relational schema mappings, scalable and efficient in producing target solutions [10].

We first discuss the overall complexity of Algorithm 1. Lines 6 and 12 involve a set-theoretic union and, without appropriate optimization, their cost is proportional to the current number of elements in $T(G)$ in each iteration of the loop. Lines 7–8 and 13–15

can be implemented in $O(1)$ provided that Lines 6 and 12 return a pointer to the element $o := f(C.\mathrm{Id}(\bar{o})) \in T(G)$. Thus the overall complexity of Algorithm 1 on input G is:

$$O\left(t_{int} + n_c \cdot Int(G,T) \cdot |T(G)|\right) \tag{3}$$

where n_c is the total number of content constructors in T, $Int(G,T)$ and t_{int} are respectively the total size of all intermediate results $[\![P]\!]^{\bar{x}}_G$ and the overall running time for computing $[\![P]\!]^{\bar{x}}_G$, with $P(\bar{x})$ ranging over all left-hand sides of rules in T.

Thus, the total time taken by Algorithm 1 implemented naively is quadratic in the size of the property graphs, which makes it practically unusable for large input instances. However, the complexity heavily depends on the implementation of the set-theoretic unions.

Plain implementation. In Figure 5 we showcase the result of our translation strategy for the variant T_{r} of T, presented in Figure 4. This transformation has only one rule and is translated into a single executable script. For transformations with several rules, each rule of the transformation is independently translated into a script.

Cypher's built-in `elementId` primitive provides access to the identifier of an element, which is unique among all elements in the database. It plays a crucial role in our implementation as we actively use these identifiers as arguments to the Skolem function generating output identifiers. To the best of our knowledge, there is no explicit control of the creation of new identifiers in Neo4j, so we equip nodes and edges in the output graph with a special property `_id` that plays the role of controllable element identifier.

Lines 1–3 correspond to the left part of the rule and are responsible for retrieving the necessary information from the input property graph. Recall that, in Line 3 of Algorithm 1, a node rule is added for each endpoint of every edge constructor in the transformation. Accordingly, in the openCypher script, each node constructor used on the right-hand side of the rule is considered separately (Lines 4–12). Similarly to how Skolem functions are usually implemented in relational data exchange for schema mapping tasks [10], we implement them with string operations, e.g., `_id: "(" + elementId(u) + ")"`. We rely on the semantics of Cypher's **`MERGE`** clause, described in [25], to implement the set-theoretic union: in Lines 4, 7, and 10, **`MERGE`** checks whether an element with this identifier already exists in the graph; either one exists and is retrieved, or a new element is created. Adding the corresponding label(s) to the retrieved node (Line 7 of Algorithm 1) is implemented with the native Cypher's **`SET`** clause in Lines 5, 8, and 11. Similarly, the properties of the nodes (Line 8 of Algorithm 1) are set in Lines 6, 9, and 12.

Finally, the relationships are created (Lines 13–18). To keep the value of `_id` unique among all elements in the output, and given the restriction that relationships hold a single label in Neo4j, the edge labels have been provided as arguments to the Skolem functions in Figure 4. Note that, when we merge an edge pattern, we are sure that the endpoints already exist in the database.

We point out that the `_id` property and the `_dummy` label are internal data; they are of no interest to the end user and can be dropped after the transformation with Cypher's **`REMOVE`** command.

Optimizations. Optimizing the **`MERGE`** clauses in Lines 4, 7, 10, 13, and 16 which implement the set-theoretic unions is crucial in reducing the overall execution time of the transformation.

As is the case in most database management systems, Neo4j provides facilities for query optimization. The two that are relevant in this context are indexes and uniqueness constraints. An *index* permits to retrieve efficiently nodes with a given label that have a specific value at a given property. When we know in advance that all these values are unique, we can make further use of *uniqueness constraints* (UCs). Note that in our implementation, we maintain the invariant that each `_id` is unique across all elements in the output.

In the version of Neo4j Community Edition that we use for running the experiments, indexes are implemented using b-trees, which means that the cost of testing if an index with n elements contains a given key is $O(\log n)$. That is, by using indexes we can improve the worst-case complexity of Algorithm 1 to:

$$O\left(t_{int} + n_c \cdot Int(G,T) \cdot \log |T(G)|\right) \tag{4}$$

In the next section we comprehensively evaluate the advantages and disadvantages of using indexes and uniqueness constraints on nodes and relationships, defined on the label/property pair `_dummy`/`_id`.

Conflict detection. The consistency problem is unfortunately PSpace-complete by Corollary 4.7, so we cannot efficiently check the declarative specification at compile time. Instead, we need to be ready for potential inconsistencies at run time.

Figure 6 illustrates how one can detect conflicts on the property `code` when creating a new `City` node. We use the **`ON MATCH`** subclause of the **`MERGE`** clause to perform a comparison when we set a property for an existing node. Notice that a different rule could have led to the creation of this node and, consequently, `z.code` may be empty; in this case the operator <> returns **`false`** and the correct specification is reached.

6 EXPERIMENTS

Our experimental study has three main objectives: (i) evaluate the benefits of using this formalism for transforming property graphs in practical use-cases over a large amount of data, (ii) evaluate the involved overhead of detecting potential inconsistencies at runtime, and (iii) compare with the native openCypher approach such as the one presented in Figure 1 (ii).

Experimental setting. We have implemented our property graph transformations in openCypher 9 using a local Neo4j Community Edition instance in version 5.9.0. For monitoring the results and performing the database management tasks required in our methodology, we have used Python 3.11 and the official Neo4j Python Driver 5.9.0. The source code, datasets, and configuration files are available on the public GitHub repository of the project. We performed the experiments on an HP EliteBook 840 G3 with an Intel Core i7-6600U CPU and 32GiB of system memory (2133 MHz).

Datasets. Due to the lack of benchmarks for property graph transformations, in order to build realistic scenarios we have adapted the mappings from several relational data integration scenarios from the iBench suite [4]. In particular, we encode relational input instances as property graphs by creating a node for each tuple (no edges), and we let the target instances be property graphs as well, thus simulating graph-to-graph transformations. Each mapping in a scenario corresponds to a rule of our formalism. Following the method described in Section 5, we compute an openCypher script implementing each rule.

$$(u:\text{User}),(a:\text{Address}),(\ell:\text{Location}) \underset{\langle u.address=a.aid,\, u.address=\ell.aid\rangle}{\Longrightarrow} (x=(u):\text{Person}) \xrightarrow[\langle name=u.name\rangle]{\text{(HasLocation)}:\text{HasLocation}} ((\ell.countryName):\text{Country}) \underset{\langle name=\ell.countryName,\, code=\ell.countryCode\rangle}{},$$

$$(x) \xrightarrow[\langle name=a.cityName,\, code=a.cityCode\rangle]{\text{(HasAddress)}:\text{HasAddress}} ((a.cityName):\text{City})$$

Figure 4: Refined property graph transformation T_r.

```
MATCH (u:User)
MATCH (a:Address) WHERE a.aid = u.address
MATCH (l:Location) WHERE l.aid = u.address
MERGE (x:_dummy { _id: "(" + elementId(u) + ")" })
SET x:Person,
    x.name = u.name
MERGE (y:_dummy { _id: "(" + l.countryName + ")" })
SET y:Country,
    y.name = l.countryName, y.code = l.countryCode
MERGE (z:_dummy { _id: "(" + a.cityName + ")" })
SET z:City,
    z.name = a.cityName, z.code = a.cityCode
MERGE (x)-[hl:HasLocation {
    _id: "(" + elementId(x) + "," + "HasLocation" + ","
             + elementId(y) + ")" }]->(y)
MERGE (x)-[ha:HasAddress {
    _id: "(" + elementId(x) + "," + "HasAddress" + ","
             + elementId(z) + ")" }]->(z)
```

Figure 5: openCypher script corresponding to T_r (Figure 4).

```
MERGE (z:_dummy { _id: "(" + a.cityName + ")" })
ON CREATE SET z:City, z.code = a.cityCode
ON MATCH SET z:City, z.code = CASE WHEN z.code <> a.cityCode
             THEN "Conflict detected!" ELSE a.cityCode END
```

Figure 6: Detecting conflicts on the property code.

Table 1: Scenarios characteristics.

Scenario	Labels / Properties $\lvert\mathcal{L}_{in}\rvert$	$\lvert\mathcal{L}^{node}_{out}\rvert$	$\lvert\mathcal{L}^{edge}_{out}\rvert$	$\lvert\mathcal{K}\rvert$	Rules $\lvert T\rvert$	n_c
PersonAddress	2	2	1	7	2	6
FlightHotel	2	3	2	5	1	7
PersonData	3	3	2	3	1	5
GUSToBIOSQL	7	5	4	80	8	18
DBLPToAmalgam1	7	5	4	140	10	22
Amalgam1ToAmalgam3	15	2	1	128	8	22

The middle part of Table 1 reports the number $|\mathcal{L}_{in}|$ of input labels in each scenario (corresponding to the number of different relations in the original iBench scenario), the number $|\mathcal{L}^{node}_{out}|$ of output node labels, the number $|\mathcal{L}^{edge}_{out}|$ of output edge labels, and the number $|\mathcal{K}|$ of properties. The right part provides information about the number of rules in the scenario $|T|$ and the total number n_c of content constructors. In each scenario, for each of the $|\mathcal{L}_{in}|$ input node labels, we generated up to 10^5 nodes.

Methodology. The main abstraction in our implementation is a *Scenario* which describes an input property graph database that contains some data of a given size stored in specific node and relationship properties. As previously shown in Figure 1 (i), given the iBench output, we create a node for each tuple, having as properties (key/value pairs) the columns names and column values. We also add the Cypher specification of a set of indexes and constraints on the output side, that are created before executing the transformation

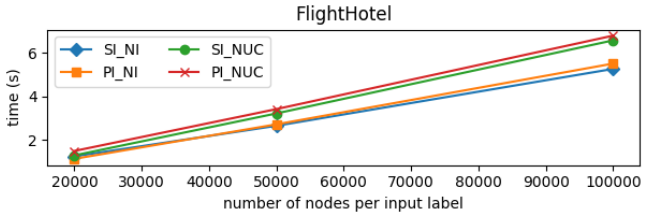


Figure 7: Comparison between uniqueness constraints and indexes for computing $T(G)$.

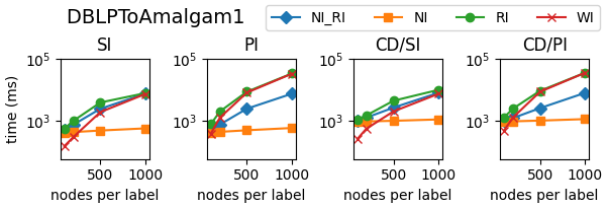


Figure 8: Impact of indexing strategies and implementation variants on the computation of $T(G)$.

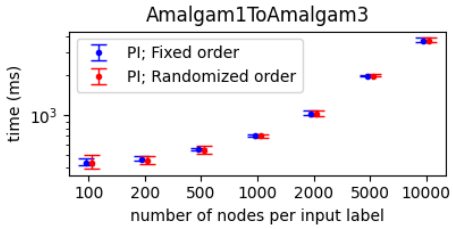


Figure 9: Average computation time for different orders of execution of the rules.

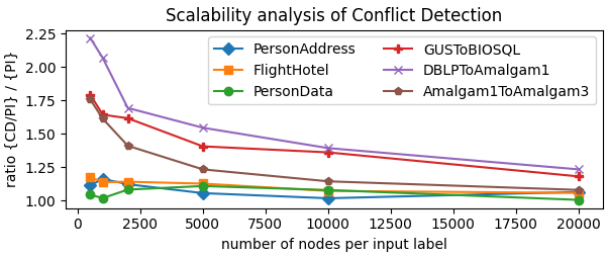


Figure 10: Ratio between the time for computing $T(G)$ with and without conflict detection (using PI_NI).

when the output data is still empty. This step is not time consuming and takes on average less than one millisecond per index.

A scenario includes several Cypher queries—one for each transformation rule—that are successively applied. To simulate the process of transforming one graph into another, and to distinguish between input and output data, we have used disjoint sets of labels in the input and output instance. Thus, a single database instance holds both input and output data at a time, but contains initially no output data. As a final step, a scenario is responsible for flushing the database and removing the indexes and constraints in order to have a fresh database instance before executing the next scenario. Note that the query cache (execution plans) is cleared when one of them is dropped. We monitored the total amount of time spend by Neo4j in applying the transformation rules. Each experiment generally represents the average taken over 5 runs of a scenario.

Alternative implementation using separate indexes. In Section 5, we discussed an implementation of the framework, the *Plain implementation* (PI), which uses a single index on the output side to speed up the retrieval of already existing nodes by Cypher's `MERGE` clause. Using a single index for all nodes in the output may severely impact the performance of the implementation as the cost of index maintenance may become prohibitive. To quantify this, we compare with an alternative implementation, the *Separate indexes implementation* (SI), where the label is part of the argument list, similar to the case of relationships. The goal here is to mitigate the cost of maintaining a very large index by splitting the data into many smaller ones. Note that it is still possible to detect conflicts in this variant with a slight modification of the code from Figure 6.

Impact of indexes and uniqueness constraints. We start by comparing the advantages of using uniqueness constraints on nodes (NUC) and indexes (NI) on the two alternative implementations, NI and SI. Figure 7 reports the results for our FlightHotel scenario, showing that for large input data, indexes tend to outperform UCs.

We next investigate the impact of using combinations of indexes on nodes and relationships. We compared variants with *indexes on nodes and relationships* (NI_RI), *indexes on nodes only* (NI), *indexes on relationships only* (RI), and *without indexes* (WI) for the previous PI and SI implementations and their respective variants with conflict detection enabled: *Conflict Detection over Plain implementation* (CD/PI), *Conflict Detection over Separate indexes* (CD/SI). We showcase in Figure 8, on a logarithmic scale, the results that were obtained for the DBLPToAmalgam1 scenario. Other scenarios show similar trends and they are reported in the extended version of our paper [13]. It is clear from the figure that the choice of indexes to use is crucial. Using indexes only on nodes is more efficient than using a combination of indexes on nodes and relationships, which is in turn more efficient than using indexes only on relationships or using no index at all. The key reason of this behavior is that indexes on nodes already allow accessing the endpoints of edges, along with the edges themselves, efficiently. Additional indexes on edges do not help, but do incur additional overhead.

The positive point that emerges from this study is that the implementation does not require fine tuning to be efficient in a specific scenario; using indexes only on nodes is consistently the best approach to use. Additionally, when using indexes only on nodes (NI), the Plain implementation (PI) is negligibly slower than Separate indexes implementation (SI), whereas for other combinations of indexes it is noticeably slower. We discussed in Example 3.4 that PI allows for more flexible use of labels compared to the SI (which corresponds to having a dedicated Skolem function for each set of labels). In view of the above results, in the remaining experiments we focus on the Plain implementation with node indexes (PI_NI).

Impact of rule order. Our formalism is declarative and does not specify the order for the execution of the rules. Hence, we have investigated the impact of different orders on the computation time of the transformation. We compare the minimum, average and maximum running times using random orders with the (fixed) order provided in iBench as baseline. Figure 9 reports the results for the DBLPToAmalgam1 scenario; error bars indicate minimum and maximum computation times observed over 20 independent runs. For space reasons, GUSToBIOSQL and Amalgam1ToAmalgam3, exhibiting similar results, are deferred to the extended version [13].

We can observe that the impact of the order in which the rules are applied on the execution time of the transformation is not substantial; randomized orders have a variance similar to that of a fixed order. It is fair to say that the performance of our implementation does not rely on any specific execution order.

Overhead of detecting potential inconsistencies. We evaluated the impact of turning on conflict detection (over PI_NI) by investigating the ratio between computation time with and without conflict detection. The theoretical complexity of our implementation of Algorithm 1 with conflict detection is:

$$O\left(t_{int} + n_c \cdot Int(G,T) \cdot (\log|T(G)| + c)\right) \tag{5}$$

for c a constant modeling the cost of the conditional statement. Thus the overhead incurred by detecting conflicts is $1 + \frac{c}{\log|T(G)|}$, which tends to 1 in larger scenarios.

The results presented in Figure 10 experimentally validate that the incurred overhead of conflict detection is reasonably low for large input instances, and stays within a constant factor, roughly between 1 and 1.3, depending on the scenario.

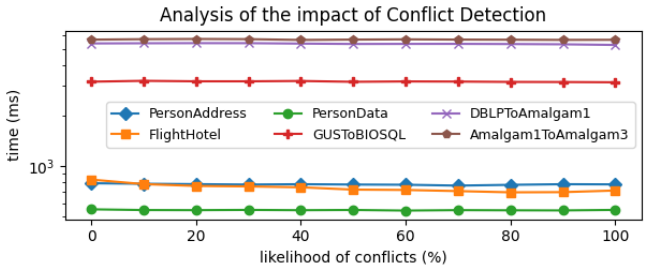


Figure 11: Run-time comparison for different likelihood of conflicts (using CD/PI_NI with 10^5 nodes for each input type).

Robustness against incidence of conflicts. iBench's scenarios have very few or no conflicts. To investigate the generalizability of these results to more conflict-prone scenarios, we designed an experiment using an additional randomization step: when a rule attempts to set a value for an attribute, the value is changed randomly. This allows us to control the average number of conflicts in the output. Figure 11 reports, on a logarithmic scale, the results for all our scenarios, with varying likelihood of conflicts, ranging from 0% to 100%. Note that, the size of the output is preserved because only the attributes are affected, not the topology of the graph.

Table 2: Running times and size of intermediate data (ICIJ).

Rules	t_{int}	t	$Int(G,T)$	$\|T(G)\|$	O/I	t^e_{int}	t^e
$R1-R4$	2,757	11,192	374,955	748,524	1.996	0.007	0.015
$R5-R9$	3,553	5,946	62,242	82,616	1.327	0.057	0.072
$R10-R13$	15,509	36,775	1,906,686	1,905,547	0.999	0.008	0.019
$R14-R17$	9,667	21,006	493,556	1,173,720	2.378	0.020	0.018
$R18$ only	8,407	25,640	785,124	1,570,470	2.000	0.011	0.016

We observe that the prevalence of conflicts has no impact on the execution time, suggesting that our framework's stability is preserved, even with a large proportion of conflicts in the output.

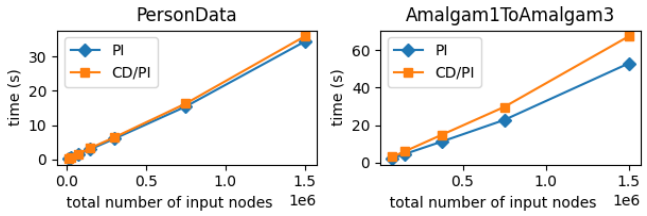


Figure 12: Horizontal scaling, with varying number of independent copies of the scenario.

Horizontal scalability. We have investigated how well our framework scales with the number of rules and input labels. We built larger scenarios by taking an increasing number of independent copies of the scenarios from Table 1. The resulting transformations reach over one hundred rules and input labels, and over 1.5 million input nodes (in total). Figure 12 reports the results for the PersonData and Amalgam1ToAmalgam3 scenarios.

We observe the running time scales smoothly (almost linearly) as the number of copies increases. Results on other scenarios follow similar trends and are deferred to the extended version [13].

Improvement over handcrafted scripts: a user study. To compare empirically the readability and usability of the script-based approach and our framework, we ran an ad-hoc user study involving 12 participants that were all already familiar with openCypher.

We compared the ability of the participants to understand the behavior of some provided openCypher scripts and transformations in clearly defined scenarios. Only 25% of the participants have been able to fully understand the behaviour of the openCypher scripts, whereas 67% of them succeeded with transformations. In average, participants have scored 50% on openCypher scripts and 90% on our framework. Participants were also asked to compare openCypher scripts and our framework in terms of understandability, intuitiveness, and flexibility; they all have favored our framework by a great margin. For space reasons, the questionnaire, the participant's answers, and the full discussion of the obtained results are deferred to the extended version of our paper [13].

6.1 Use-case Study: Improving Data Integration

In this section, we want to compare the cost of running the whole transformation compared to the cost of querying the source property graph to extract the bindings (intermediate data). To this end, we use a real-world dataset, the *Offshore Leaks Database and guide from the International Consortium of Investigative Journalists (ICIJ)* [19], a property graph with 1,908,466 nodes and 3,193,390 edges taken from [32]. This dataset consolidates data from several leaks (*Panama Papers*, *Bahamas Leaks*, etc.) collected by ICIJ over a period of ten years, but still presents the consolidated data in a heterogeneous manner. The dataset contains information about *entities* (off-shore companies), *officers* of those, *intermediaries* (middlemen who help set up off-shore companies), and *jurisdictions* (countries or territories where off-shore companies are registered). We have designed a modular 18-rule transformation aiming to uniformize the presentation of the information contained in the graph. The rules are grouped into 5 subsets, each addressing a specific refactoring goal motivated below. For space reasons, we have deferred the rules themselves to the extended version of our paper [13].

Refactoring registered addresses ($R1-R4$). The ICIJ database contains the registered *addresses* of the *officers*, and *entities*. These rules are responsible for creating nodes representing *countries* and linking to the *addresses*. Because the data is semi-structured and collected from multiple sources, information may be stored in attributes that have different names, or may even not be available at all. All these cases are covered by these four rules.

Uniformizing address information for intermediaries ($R5-R9$). After careful investigation, we found that the registered *address* of *intermediaries* can be stored in three different ways in the database: (i) an *intermediary* can have a direct relationship with an *address*, (ii) the *address* can be stored in the properties of the node itself, and (iii) when neither of the two previous cases applies, it is necessary to retrieve the *address* of an *entity* linked to this *intermediary*. These rules permit to consistently store address information.

Exporting the nodes ($R10-R13$). These rules copy the node information from the source to the target; they are necessary to preserve all the information from the original graph.

Improving similarity detection ($R14-R17$). Because the dataset consolidates multiple leaks, certain specific relationships, such as *similar* and *same_as*, are used to indicate that some *officers* (resp. *addresses*) are likely to represent the same real life entity. These rules focus on exporting this data and improving the similarity detection. This is illustrated by Rule 15 shown in Figure 13 which composes the relationships *similar* and *same_as* to ensure that both its endpoints correspond to *officers* having the same address. (This is because *similar* encompasses address similarity.) Then, it checks whether their names are also similar. If both conditions hold, it safely adds a similarity edge between the endpoints in the output.

Refactoring jurisdictions ($R18$). The last rule is responsible for connecting the *jurisdictions* with their associated *countries*; this information is not explicitly stored in the initial database.

Results. Our experimental results are reported in Table 2. We report the time t_{int} (in *ms*) the database takes to retrieve the intermediate data; the total time t of running the transformation (extracting the bindings and constructing the output); the size $Int(G,T)$ of intermediate data; the size of the output $T(G)$; the ratio O/I of the size of the output to the size of intermediate data. To account for the differences in the sizes of the outputs of the respective tasks, we also report the average time t^e_{int} taken to produce each binding of the intermediate result, and the average time t^e taken to construct each element of the output. We break down the reported values into groups of rules corresponding to the aforementioned integration tasks.

There are several things that we can learn from Table 2. First, the overhead t^e/t^e_{int} of turning the intermediate results into a proper property graph is reasonable. For Rules $R14-R17$, it is even comparatively more efficient to compute the output property graph

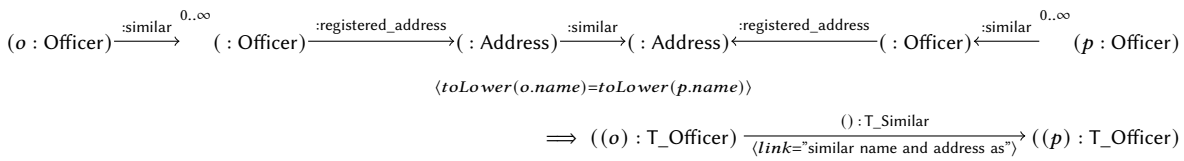


Figure 13: Improved similarity detection ($R15$).

(overhead is 0.9). The worst case is for rules $R10 - R13$ exhibiting an overhead of 2.4. Second, the ratio O/I is also reasonable, ranging from 1 to 2.4. This shows that, in practical contexts, $Int(G, T)$ can be assumed to have a size comparable to $|T(G)|$.

Thus, we have demonstrated that the overhead incurred by producing a property graph rather than a set of bindings is acceptable for a realistic transformation in a real-life integration scenario.

7 RELATED WORK

Schema mapping and data exchange. Specifying the relationship between two relational (or XML) schemas using a set of declarative assertions is a task known as *schema mapping* [9, 16, 27]. This relation, is usually *non functional*, i.e. given an input instance I, several target instances satisfying the mapping constraints exist.

Schema mappings and data exchange have been studied in [6, 23] for graph databases. The mapping languages considered are based on classical graph database queries such as regular path queries [7], limited in their expressivity by not supporting data values. Moreover, answering queries on the target is already intractable in data complexity for RPQs [6] and undecidable for data RPQs [23]. In comparison, our transformation framework provides more flexibility by including the support for data values, and any target query can be answered by simple execution on the produced property graph.

Graph generating dependencies have been introduced and studied in [30, 31]. They help specify the creation of new graph elements, hence could lead to a semantics for specifying exchange of graph data. Nevertheless, this would still provide a non functional approach, much like the above-mentioned data exchange framework.

Graph transformations. Graph database transformations based on acyclic conjunctive two-way RPQs have been investigated in [11].

The graph database model they consider does not have data values. However, we have seen that dealing with data values gives rise to the consistency checking problem, which is key to understanding if a property graph transformation is well-defined. Another difference is that they are using a single dedicated node constructor for each label. In Section 3 and 5, we have seen that this approach is too rigid for dealing with multiple labels.

Object-creating functions. The Skolem functions we use in our constructors resemble to the object creating functions that are used in the object-oriented database model [1, 26]. Among transformation languages based on oid generation, StruQL [18] specifically operates on object-oriented semi-structured instances. The major difference with our work is that they have *multi-valued* attributes. Hence, additional integrity constraints are necessary to ensure a correct modeling of property graphs in their model. Therefore, they did not take into account the problem of consistency.

Interoperability of graph data. Although RDF, RDF-star and the property graph data model share striking similarities, both being based on elementary graph concepts, like nodes and edges, intricate interoperability issues arise when attempting to exchange data between them. RDF-star notably allows for annotating RDF triples with metadata annotations, which are notoriously difficult to capture within the property graph data model as witnessed in [2].

The main concern of transformation languages between graph data models is thus primarily focused on solving the well-known impedance mismatch problem [10], which does not arise in our setting because we have property graphs for both input and output. Our transformation language can be thus more expressive, and can be executed by the graph database management system itself.

Mining the identities of nodes across networks. Network alignment is a technique for finding node correspondences between two or more networks. It can be used, for example, to associate nodes from different social networks with the same user [35]. Nodes are identified based on their similarities with respect to both their features (i.e., their properties) and their neighborhood.

While these methods are not part of graph transformation formalisms, they can be used to guide the construction of graph transformations. For instance, in Section 6.1, the results of network alignment (the similarity edges in the Offshore Leaks Database) were leveraged to better integrate data coming from multiple leaks.

8 CONCLUSION

Our research is the first to lay the theoretical foundations for declarative property graph transformations, and facilitate practical solutions for turning such specifications into executable scripts in modern property graph query languages. New challenges arise from the specification of property-aware transformations, notably the task of checking if a transformation is *consistent*. Using a proof-of-concept implementation of our formalism in openCypher, we showcase the efficiency of our approach for transforming property graphs for both real-world and synthetic datasets.

This work paves the way for obtaining compositional semantics for graph query languages. As a future direction, we will investigate the model extensions needed for the above semantics, by addressing label and path variables, and aggregates. Meanwhile, our framework can already seamlessly support the group variables of GPC because those are list of identifiers that can be flattened into the identifier lists of the constructors. Finally, we will investigate how to assist users in the design process of their transformation rules; for instance by lifting *schema matching* techniques [9, 10] from relational to property graph schemas.

ACKNOWLEDGMENTS

Angela Bonifati and Yann Ramusat were supported by the VeriGraph (ANR-21-CE48-0015) project. Filip Murlak was supported by Poland's NCN grant 2018/30/E/ST6/00042.

REFERENCES

[1] Serge Abiteboul and Paris C. Kanellakis. 1998. Object Identity as a Query Language Primitive. *J. ACM* 45, 5 (1998), 798–842.

[2] Ghadeer Abuoda, Daniele Dell'Aglio, Arthur Keen, and Katja Hose. 2022. Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches. In *QuWeDa 2022*. 17–32.

[3] Marcelo Arenas, Pablo Barcelo, Leonid Libkin, and Filip Murlak. 2010. *Relational and XML Data Exchange* (1st ed.). Morgan and Claypool Publishers.

[4] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The IBench Integration Metadata Generator. *VLDB* 9, 3 (2015), 108–119.

[5] Patricia C. Arocena, Boris Glavic, and Renee J. Miller. 2013. Value Invention in Data Exchange. In *SIGMOD*. 157–168.

[6] Pablo Barceló, Jorge Pérez, and Juan Reutter. 2013. Schema Mappings and Data Exchange for Graph Databases. In *ICDT*. 189–200.

[7] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. 2012. Relative Expressiveness of Nested Regular Expressions. In *AMW*. 180–195.

[8] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *PODS*. 175–188.

[9] Z. Bellahsene, A. Bonifati, and E. Rahm. 2011. *Schema Matching and Mapping*.

[10] Philip A. Bernstein and Sergey Melnik. 2007. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*. 1–12.

[11] Iovka Boneva, Benoît Groz, Jan Hidders, Filip Murlak, and Slawek Staworko. 2023. Static Analysis of Graph Database Transformations. In *PODS*. 251–261.

[12] Angela Bonifati, Ugo Comignani, Emmanuel Coquery, and Romuald Thion. 2017. Interactive Mapping Specification with Exemplar Tuples. In *SIGMOD*. 667–682.

[13] Angela Bonifati, Filip Murlak, and Yann Ramusat. 2024. Transforming Property Graphs. arXiv:2406.13062 [cs.DB] https://arxiv.org/abs/2406.13062

[14] Laura Chiticariu and Wang-Chiew Tan. 2006. Debugging schema mappings with routes. In *PVLDB*. 79–90.

[15] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. 2246–2258.

[16] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data Exchange: Semantics and Query Answering. *TCS* 336, 1 (2005), 89–124.

[17] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. 2005. Data Exchange: Getting to the Core. *TODS* 30, 1 (2005), 174–210.

[18] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. 1997. A Query Language for a Web-Site Management System. *SIGMOD* 26, 3 (1997), 4–11.

[19] Miguel Fiandor and Michael Hunger. [n.d.]. Offshoreleaks Data Packages. Retrieved March 1, 2024 from https://github.com/ICIJ/offshoreleaks-data-packages

[20] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A Pattern Calculus for Property Graphs. In *PODS*. 241–250.

[21] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. A Researcher's Digest of GQL. In *ICDT*, Vol. 255. 1–22.

[22] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. 1433–1445.

[23] Nadime Francis and Leonid Libkin. 2017. Schema Mappings for Data Graphs. In *PODS'17*. 389–401.

[24] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[25] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating graph databases with Cypher. *VLDB* 12, 12 (2019), 2242–2254.

[26] Richard Hull and Masatoshi Yoshikawa. 1990. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*. 455–468.

[27] Phokion G. Kolaitis. 2005. Schema Mappings, Data Exchange, and Metadata Management. In *PODS*. 61–75.

[28] Neo4j. 2023. APOC user guide for Neo4j 5. Retrieved November 9, 2023 from https://neo4j.com/docs/apoc/current/

[29] Neo4j. 2023. Graph Data Modeling Fundamentals. Retrieved November 9, 2023 from https://graphacademy.neo4j.com/courses/modeling-fundamentals/

[30] Larissa C. Shimomura, George Fletcher, and Nikolay Yakovets. 2020. GGDs: Graph Generating Dependencies. In *CIKM*. 2217–2220.

[31] Larissa C. Shimomura, Nikolay Yakovets, and George Fletcher. 2022. Reasoning on Property Graphs with Graph Generating Dependencies. arXiv:2211.00387 [cs.DB] https://arxiv.org/abs/2211.00387

[32] Philipp Skavantzos and Sebastian Link. 2023. Normalizing Property Graphs. *Proc. VLDB Endow.* 16, 11 (2023), 3031–3043.

[33] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *GRADES*. 1–6.

[34] Moshe Y. Vardi. 2016. A Theory of Regular Queries. In *PODS*. 1–9.

[35] Si Zhang and Hanghang Tong. 2020. Network Alignment: Recent Advances and Future Directions. In *CIKM*. 3521–3522.