Versatile Property Graph Transformations

Angela Bonifati
Lyon1 University, CNRS Liris & IUF
Lyon, France
angela.bonifati@univ-lyon1.fr

ABSTRACT

Property graphs are key components of modern graph database systems as well as graph analytical systems. They support highly expressive data models consisting of multi-labeled nodes and edges, along with properties represented as key/value pairs. Property graphs serve as versatile data integration paradigms, enabling data in any format to be seamlessly transformed into this model. Moreover, they are at the core of an active standardization effort led by ISO/IEC, which aims to establish standardized declarative graph query languages such as GQL and SQL/PGQ. In addition to these standards for data manipulation languages, other languages have emerged for property graph schemas and constraints as part of future data definition languages.

In this paper, we introduce a new declarative paradigm for expressing property graph transformations, supporting both graph data integration and data cleaning tasks. We discuss the properties of these transformations, along with algorithmic issues and considerations for efficiency and scalability. Furthermore, we showcase the utility of property graph transformations for causal analysis and elaborate on a research agenda aimed at designing analytical extensions of graph languages to support property graph transformations for advanced analytical workloads on heterogeneous data.

PVLDB Reference Format:
Angela Bonifati. Versatile Property Graph Transformations. PVLDB, 18(12): 5516-5526, 2025.
doi:10.14778/3750601.3760517

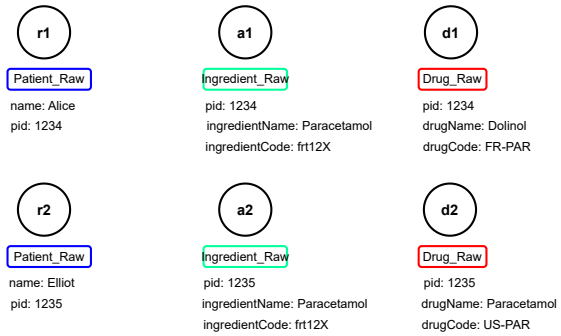
1 INTRODUCTION

Property graphs (PGs) are highly expressive graph data models underlying commercial and open-source native graph database systems and providing graph extensions of relational engines. Systems such as Neo4j [3], Amazon Neptune [1], Oracle PGX [4], SAP Hana Graph [6], RedisGraph [5], Sparksee [7], Kuzu [2], Google Cloud's Spanner Graph [32], MilleniumDB [80] and others have been flourishing in the last few years, leading to the need of unification and standardization of graph query languages and data models. This unification was urgently needed to pave the way to Big Graph ecosystems integrating different data model abstractions [70]. Standardization efforts led by ISO/IEC have worked on key graph data models and query language standards, such as GQL and SQL/PGQ [11, 34, 43], respectively providing standardized native and SQL-based graph query languages. As additional endeavors, schema and constraint languages for property graphs have been studied and formalized, including PG-Schema [9], a schema language for property graphs, whose constructs have influenced the standard query language schema support. The design of these languages started from a set of possible desiderata, on which both academia and industry agreed [10], spanning from property graph types (node types, edge types and property types) to constraints (PG-keys and PG-constraints such as participation constraints and node hierarchies), along with flexibility (such as compositionality and evolution support), and usability (property graph schema generation, efficient schema constraint validation, well-defined syntax and semantics).

PG-Schema and PG-Constraints anticipate the need of data integration and quality enforcement for property graphs. Data integration [16, 36, 56, 61, 77] is a workhorse of data management research with several influential papers on the topic. Integrating property graphs poses new challenges due to the fact that query discovery [61] leads to graph transformations that are more complex than relational data. In particular, in order to transform input data in any format (e.g. in relational format—see Example 1.1) into an output property graph, node, edge and subgraph creation is needed to encode the query behind the transformation itself. There are plenty of relevant applications, which require a graph view on top of both structured and unstructured data, such as for instance lakehouses and graph retrieval augmented generation (RAG) to enhance the accuracy of Large Language Models (LLMs). However, the graph transformation process can be harnessed by conflicting values thus showing the need of precise conflict detection mechanisms. Furthermore, property graph transformations can be seen as a mechanism to define declarative graph views, which is an operation lacking in current standardized graph query languages. Graph views are essential for the efficient evaluation of graph queries and for enabling data integration scenarios in the graph world [24, 47].

A further usage of property graph transformations is in the handling of violations of graph constraints. Graph constraints can be expressed declaratively as PG-Keys and as graph denial constraints. In both cases, repairs to violation are derived based on some criteria, such as cost or user input. Previous approaches have leveraged preferred repairs [76] or user-centric repairs [63, 64] (see Example 1.2 for the latter). Depending on the repair model utilizing insertions and/or deletions of properties, nodes, edges and subgraphs, or relabelings of nodes, the output of a repair can be seen as a graph transformation in which the violations are fixed.

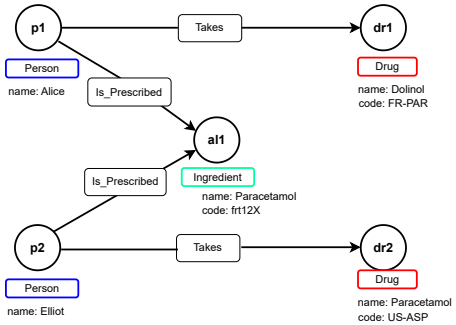
Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3760517



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

```
MATCH (p_raw:PatientRaw)
MATCH (i_raw:IngredientRaw) WHERE i_raw.pid = p_raw.pid
MATCH (d_raw:DrugRaw) WHERE d_raw.pid = p_raw.pid
WITH p_raw, collect(i_raw) AS Ingredients,
            collect(d_raw) AS Drugs
CREATE (p:Person)
SET p.name = p_raw.name
WITH p, i_raw
UNWIND i_raw.ingredients AS ingredientData
MERGE (i:Ingredient {name: ingredientData.name})
SET i.code = ingredientData.code
MERGE (p)-[:IS_PRESCRIBED]->(i)
WITH p, d_raw
UNWIND d_raw.drugs AS drugData
MERGE (d:Drug {name: drugData.name})
SET d.code = drugData.code
MERGE (p)-[:TAKES]->(d)
```

(ii) Ad-hoc transformation script in openCypher.



(iii) Resulting output property graph H.

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

1.1 Property Graph Transformations for Versatile Data Manipulation

In this paper, we introduce a new data manipulation paradigm based on declarative property graph transformations for data integration and data cleaning. We discuss the declarative nature of these transformations as well as the algorithmic issues along with the efficiency and scalability desiderata. We will not assume that all the data is already encoded as property graphs as we show that graph transformations can be used across different data models (e.g. from relational to graphs). Our work also shows that graph transformations are versatile and can be used in other data science applications, for instance to express data manipulation operations on causal artifacts represented as causal DAGs (Directed Acyclic Graphs). Causal analysis is one of the future endeavors in Artificial Intelligence where graph transformations are useful to encode causal interventions. Intervening on a variable means forcing all the units of the population to take a specific value for that variable, independently on its true causes, and observing the changes in the other variables accordingly. If causal DAGs are treated as database objects, these causal operations can be modeled as graph transformations. In the following, we present two examples of applications of graph transformations in the medical domain. The first example shows property graph transformations at work when restructuring medical data across different formats. The second deals with property graph transformations induced by repairing violations in a medical property graph.

1.2 Graph Transformation Examples

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

$$\text{Patient\_Raw}(\underline{name}, \underline{pid}),$$
$$\text{Ingredient\_Raw}(\underline{pid}, ingredientName, ingredientCode),$$
$$\text{Drug\_Raw}(\underline{pid}, drugName, drugCode),$$

with primary keys consisting of the underlined attributes, and having two foreign keys: *pid* in Patient references *pid* in Ingredient and pid in Drug.

Figure 1(i) shows a straightforward property graph obtained after importing 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 patients, ingredients, and drugs are represented by way of foreign keys, just like in the original relational instance. Needless to say, this is not an ideal translation of a relational instance into 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 have been prescribed the same ingredient as Alice?". The user intends to create a node for each patient, ingredient, and drug, and replace foreign key references with explicit relationships. Notice that the desired property graph transformation must produce a unique output property graph, integrating the three input relational tables shown above. Figure 1(ii) shows an implementation of this transformation in openCypher, a popular graph query language used in graph databases, whose recent versions incorporate novel features from standard languages such as GQL [51]. 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`

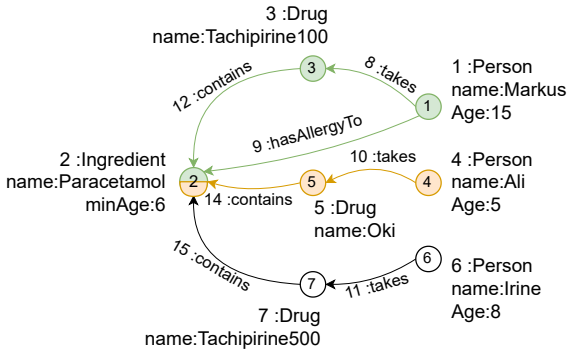
and `collect`) is a daunting task for an unacquainted user. But the query leverages other advanced idioms too. For instance, in Line 6, the script creates as many nodes of type `Person` as there are rows output by the previous **`WITH`** clause: one for each r, due to implicit grouping. In line 10, the script generates one `Ingredient` node for each *distinct* value found in property `ingredientName` across all i's; this is because the property `name` is specified as `d.ingredient` in the **`MERGE`** clause. Similarly, in line 15, a single `Drug` node is created for each *distinct* value found in property `drugName`.

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 ingredients are *weak entities* that cannot be identified by their name alone, and conflates the two ingredients into one despite the fact that they refer to different drugs with different drug codes. Detecting such errors is hard because openCypher lacks a transparent mechanism for specifying identities of created elements.

Example 1.2. The property graph depicted in Figure 2 is a slightly different property graph that exemplifies the problem of repairing data about patients and their treatments, represented with property and edge labels (such as *:Person* and *:hasAllergyTo*) and properties as key-value pairs (*name=Aspirin*). The property graph reflects a possible real-world state of a database containing patients, *Markus*, who is allergic to *Paracetamol*, and *Irine* who takes *Tachipirine500*. Plausible property graph denial constraints ϕ_1 and ϕ_2 in this example state that 1) patients cannot be treated with a drug they are allergic to, and 2) a patient cannot be treated with a drug if they are younger than the minimum age allowed for the ingredient of the drug. We show how the application of these constraints can be handled by using property graph transformations.

The subgraphs highlighted in green and orange showcase respectively two violations of the above constraints, namely (1) *Markus* is taking *Tachipirine100* that contains *Paracetamol* while being allergic to it, and (2) *Ali* taking *Oki* that contains *Paracetamol* while being too young. These two violations can be fixed applying four different graph transformations: (a) one can modify the node with name *Tachipirine100* with the one of another *Drug* (and consequentially change the ingredient)—corresponding to a property update—(b) the edge between patient and medication can be deleted, (c) the edge between the drug and the ingredient can be removed (for instance, *Oki* does not contain *Paracetamol* but contains *Ibuprofene*), (d) or the edge *hasAllergyTo* can be suppressed because the patient was not allergic to *Paracetamol*. As opposed to Example1.1, we can observe the multiplicity of solutions produced by different graph transformations, along with the use of denial constraints to operate the results of the transformations. The actual choice of which transformations to use can be guided by suitable properties of the transformations and guided by the users, as discussed in the remainder of the paper.

The paper is organized as follows. Section 2 provides an overview of standardized property graph languages. Section 3 presents the basic ingredients of property graph logic-based transformation rules and discusses their formalization and implementation. Section 4 focuses on an application of graph transformations to user-centric property graph repairs. Section 5 focuses on property graph transformations and graph views for causal analysis. Each of the sections 3, 4 and 5 exposes the problem and its solution along with a concise discussion of past work on the topic. Section 6 concludes the paper and discusses open research challenges in the area.



$$Q_1(x_1, y_1, z_1, z_2) \leftarrow :hasAllergyTo(x_1, z_2) \wedge :takes(x_1, y_1)$$
$$\wedge :hasIngredients(y_1, z_1)$$
$$\phi_1 = (Q_1(x_1, y_1, z_1, z_2), id(z_1) = id(z_2) \rightarrow \bot)$$
$$Q_2(x_1, y_1, z_1) \leftarrow :takes(x_1, y_1) \wedge :contains(y_1, z_1)$$
$$\phi_2 = (Q_2(x_1, y_1, z_1), x_1.age < z_2.minAge \rightarrow \bot)$$

Figure 2: A property graph depicting two violations in green and yellow.

2 EVOLUTION OF PROPERTY GRAPH STANDARDS

2.1 DML for Property Graphs

A Data Manipulation Language is a part of the database query language standards dedicated to retrieval and update operations. Compared to the relational model, property graph queries have richer expressive power as they allow to perform concise graph pattern matching and navigational operations. Such graph traversal operations would require as many joins as the number of nodes traversed in each path. As a further complication, in recursive path traversal, the number of joins is unbounded.

Cypher [41] was introduced by Neo4j and later open-sourced as openCypher. It quickly became popular, due to its declarative nature and very intuitive ASCII-based syntax, allowing even non-expert users, who do not (want/need to) know about graph traversals, to easily form their queries. Cypher also supports CRUD operations (Create, Read, Update, and Delete) and is at the heart of a widely used rich property graph analytics ecosystem.

SQL/PGQ[1] is the first ISO standard property graph query language defined in late 2023 to include property graph queries in SQL. It allows read-only graph views of relational data, allowing graph query functionality (e.g., path expressions and graph traversals) within the context of SQL queries. As such, it fosters backward compatibility with relational SQL.

GQL[2], is the second ISO standard property graph query language defined at the beginning of 2024, stemming from the unification of native graph query languages, such as G-CORE [8] and openCypher. GQL supports a superset of the features offered by SQL/PGQ and is designed to offer a declarative approach to graph querying, allowing

[1] https://www.iso.org/standard/79473.html
[2] https://www.gqlstandards.org/,https://jtc1info.org/slug/gql-database-language/

users to express complex graph patterns, path queries, and data transformations in a concise and intuitive manner.

All the above languages, including the first versions of the standards GQL and SQL/PGQ [11, 34, 43] focus on returning relational tuples instead of returning property graphs. More details about these languages can be found in a recent tutorial [55].

Future versions are expected to have compositionality and to support outputs as property graphs. As such, all these languages are not yet ready to express graph transformations returning property graphs, as we further discuss in Section 3. Path-based algebraic operators leading to return paths instead of plain relational tuples along with the needed standard graph query language extensions are discussed in [11]. These operators will be of key importance to study optimization opportunities and compositionality requirements of future graph query languages.

2.2 DDL for Property Graphs

A Data Definition Language (DDL) is the part of the standard dedicated to enforce data integrity rules. In parallel with DML standards for property graphs, DDL schema design has been carried out. For the time being, some of the features of the proposed DDL are incorporated in the DML standards, but there is not yet a part of the standardized GQL and SQL/PGQ dedicated to DML.

In this area, the following approaches have been developed:

A DDL for Cypher [23] was the first extension of Cypher to include schema constraints, focusing on node types, edge types and property types in Cypher-like syntax.

PG-Keys [10] emerged from a collaborative community effort in LDBC (Linked Data Benchmark Council) and introduced flexible key constraints with different modes (combinations of exclusive, mandatory, and singleton restrictions) that can be applied to nodes, edges, and properties.

PG-Schema [9] was developed in LDBC as a comprehensive schema formalism that combines flexible type definitions supporting multi-inheritance with expressive constraints based on the PG-Keys framework.

These developments are expected to influence the second version of the GQL standard, which is expected to include a rich DDL, making graph database systems more useful, powerful, and expressive while ensuring proper data integrity and object identification capabilities.

3 DECLARATIVE PROPERTY GRAPH TRANSFORMATIONS

We present a declarative formalism for specifying property graph transformations. Transformations are encoded by means of property graph transformation rules. Each rule collects data from the input graph with Graph Pattern Calculus (GPC) pattern [39] on the left of $\Longrightarrow$, and specifies elements of the input graph using content constructors 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). New identifiers are generated using *Skolem functions* and how identifiers, labels, and properties of output elements are specified using *content constructors*. A precise semantics is defined for the transformation rules in terms of a procedure that generates an output property graph given an input property graph.

3.1 Past Approaches

Specifying the relationship between two relational (or XML) schemas using a set of declarative assertions is a schema mapping task [16, 36, 54]. This relation, is usually *non functional*, i.e. given an input instance I, several target instances satisfying the mapping constraints exist. Executable SQL scripts for relational schema mappings along with scalable and efficient in producing target solutions have been studied [17]. This solution is not directly applicable to property graphs, due to the inherent differences of DML languages.

Schema mappings and data exchange have also been studied in [13, 42] for graph databases. The mapping languages are based on classical graph database queries such as regular path queries [14], limited in their expressivity by not supporting data values.

Graph generating dependencies have been introduced and studied in [72, 73]. 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 database transformations based on acyclic conjunctive two-way RPQs have been investigated in [21]. The graph database model they consider does not have data values. Moreover, this approach is too rigid to work with multiple labels.

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. A compelling cross-model graph transformation approach [68] has leveraged SHACL to PG-Schema conversion to guide the transformation from RDF to property graphs.

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 [85]. 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. In our work [24], the results of network alignment has been showcased for the Offshore Leaks Database [62], a popular property graph dataset, in order to better integrate data coming from multiple leaks based on the similarity of the edges.

3.2 Property Graph Transformation Rules

In this subsection, we formalize property graph transformation rules starting with the definition of content constructors, which are key components of graph transformations. 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 $C(\bar{x})$.

In the first rule in Figure 3, new Ingredient nodes are described using the following content constructor:

$$\begin{aligned} C_{\text{t}}(r, a, d) \coloneqq \{ \\ \text{Id: } (d.ingredientName) \\ \text{Labels: } \{\text{Ingredient}\} \\ \text{Properties: } \langle name = d.ingredientName,\ code = d.ingredientCode \rangle \ \}. \end{aligned}$$

It specifies the identities and the values of properties *name* and *code* of new Ingredient nodes in terms of the values of properties *ingredientName* and *ingredientCode* retrieved from elements to which variable d is bound in the input graph. Rather then using the abstract syntax introduced above, the rule in Figure 3 presents C_{t} in GPC-like syntax [40] as

$$\underset{\langle name=d.ingredientName,\ code=d.ingredientCode \rangle}{((d.ingredientName) : \text{Ingredient})} .$$

A *property graph transformation* is a finite set 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.

The semantics of GPC is defined such that a query returns *tuples* in line with concrete graph query languages, such as GQL, openCypher and SQL/PGQ. 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:

$$P(\bar{x}) \implies (C_{\text{s}}(\bar{x})) \xrightarrow{C(\bar{x})} (C_{\text{t}}(\bar{x}))$$

where $P(\bar{x})$ is a GPC query with singleton variables $\bar{x}$ and $C_{\text{s}}(\bar{x})$, $C(\bar{x})$ and $C_{\text{t}}(\bar{x})$ are content constructors. Node rules and edge rules can be composed to form rules that create subgraphs of arbitrary complexity. An example of property graph transformation is shown in Figure 3 consisting of two transformation rules. The first rule is generating Person and Drug nodes connected by edges labelled with Takes, while the second generates Ingredient nodes. The identifiers of the Drug nodes and the Ingredient can be controlled by means of Skolem functions accepting as arguments the values of the properties drugName and ingredientName, respectively. The transformation might then conflate the nodes Ingredient and the node Drug when the property value is "*Paracetamol*" producing an unexpected output. Detecting the modelling error in the rules in Figure 3 requires human expertise. On the other hand, setting an output property to conflicting values can be detected automatically. In our work [25], 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 also study how to detect conflicts at runtime when static analysis is not feasible.

3.3 Executable Property Graph Transformations

The semantics of the above property graph transformation rules can then be defined and the algorithmic details can be found in [24]. The semantics specification 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. Then, one can show to compile a transformation to an openCypher script that can be directly executed in any openCypher engine. Suitable extensions to the graph query language itself, such as the introduction of a `GENERATE` clause, is also desirable [26].

The total time taken by the naive implementation of the algorithmic procedure for property graph transformations is quadratic in the size of the property graphs, which makes it unpractical for large input graph instances. However, the time complexity heavily depends on the implementation of set-theoretic unions.

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, which has been used for our implementation. Hence, we equip nodes and edges in the output graph with a special property `_id` that plays the role of controllable element identifier.

Efficient ways of running these transformation scripts have been cross-compared using off-the-shelf indexes or uniqueness constraints and their efficiency and scalability have been studied. An *index* permits to retrieve efficiently nodes with a given label that have a specific value at a given property. In practical graph data management systems, indexes are implemented using B-trees, which means that the cost of testing if an index with n elements contains a given key is $O(\log n)$, thus improving the the worst-case complexity of the transformation algorithm.

When we know in advance that all values of the properties are unique, we can make further use of *uniqueness constraints*.

The experimental results show interesting trends that can inspire future graph transformation implementations to be combined with modern graph query processing techniques [60]. We have compared the pros and cons of using uniqueness constraints on nodes and indexes on nodes and edges. The results show that for large transformation scenarios indexes tend to outperform uniqueness constraints. 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 and incur additional overhead.

$$(r:\text{Patient}),(a:\text{Ingredient}),(d:\text{Drug}) \underset{\langle r.pid{=}a.pid,\, r.pid{=}d.pid\rangle}{\Longrightarrow} ((r):\text{Person}) \underset{\langle name{=}r.name\rangle}{\xrightarrow{:\text{Takes}}} \underset{\langle name{=}d.drugName,\, code{=}d.drugCode\rangle}{((d.drugName):\text{Drug})} \quad (1)$$

$$(r:\text{Patient}),(a:\text{Ingredient}),(d:\text{Drug}) \underset{\langle r.pid{=}a.pid,\, r.pid{=}d.pid\rangle}{\Longrightarrow} ((r):\text{Person}) \underset{\langle name{=}r.name\rangle}{\xrightarrow{:\text{Is\_Prescribed}}} \underset{\langle name{=}a.ingredientName,\, code{=}a.ingredientCode\rangle}{((a.ingredientName):\text{Ingredient})} \quad (2)$$

$$(3)$$

Figure 3: Transformation T given as a set of rules.

The lesson learned is that one does not have to tune the implementation depending on the particular case since using indexes only on nodes is consistently the best approach to use.

4 PROPERTY GRAPH TRANSFORMATIONS FOR GRAPH REPAIRS

Given a property graph G and a violation I, the repair G' of G is a graph transformation in which the violation I has been fixed. Compared to other data models, repairing a violation in a property graph G might trigger new violations (e.g., in the presence of multiple labels); (ii) the same errors can be fixed in several different ways, considering that a property graph includes labels and properties on both nodes and edges (e.g., a repair on an edge property X is different from a repair on a node property with the same name).

We need to detect all the graph patterns that cause violations. As exemplified in Figure 2, the graph pattern Q_1 and Q_2 have different matches in the property graph (highlighted in green and orange). To reach this goal, we need to map the constraints into queries in order to retrieve all the violations as subgraphs.

4.1 Past Approaches

Graph repairs include various methods for cleaning data that violate integrity constraints. Multiple constraints have been proposed with different expressivity, including Graph Functional Dependencies [38], Graph Entity Dependencies and Graph Denial Constraints [37]. For what concerns the repairs, the deletion model [29] has been proposed to remove erroneous data, while the update model [82] applies transformations to rectify data and ensure compliance with constraints. In this work, we employ a hybrid approach, combining both deletion and update models. Graph database repair spans across different dependency classes [22, 37, 57]. In a prior work [75], repairs involve both deletion and update operations for neighborhood constraints, addressing conflicts through relabeling (update) and edge deletion. However, this work does not study interactive property graph repairing techniques with graph denial constraints and is limited to a special class of constraints. Other works used rules [28] to repair graphs, but they assume that the rules are known in advance, or are derived from a graph that is already error-free.

Human-in-the-loop approaches have been applied to data cleaning [30, 48, 84] to ensure better quality fixes with respect to automated repairs, or to discover constraints from dirty datasets [74, 79] Interactive graph repairing has also found application in prior work [52], particularly in the context of repairing labeled graph under neighborhood constraints. In this paper, our focus is on more expressive graph data models and constraints (i.e. property graphs and denial constraints). Moreover, for the first time, we study a user interaction model targeting multiple users.

4.2 Repairs as Property Graph Transformations

Given a property graph G and a denial constraint ϕ with graph pattern Q, such as the ones depicted in Figure 2 a subgraph I *satisfies* ϕ if there does not exist a match $h(I)$ homomorphic to I of the graph pattern Q in G. Otherwise, I is a *violation* of the constraint ϕ. Interpreting denial constraints as queries has been done before in the vast literature on data cleaning [50]. The caveat here is to consider graph queries that probe edges and nodes with multiple labels along with property values.

Given a property graph instance G and a violation I, a repair r is a sequence of property graph transformations $(\mathcal{T}_0, ..., \mathcal{T}_m)$ with $m \geq 0$ returning a property graph $G' = (V, E', \eta', \lambda', \nu')$ with no match $h(I)$ of I in G'.

In this work, we have considered three types of property graph transformations, namely:

(i) edge-deletion iff the edge set $E' \subset E$;
(ii) relabeling iff $\lambda \neq \lambda'$;
(iii) property-update iff $\nu \neq \nu'$.

We introduce the *Graph Repair Dependency Graph* (GRDG), a Hypervertex Labeled Property Graph, which models an overlay of graph violations.

It serves the need of mapping the results of graph pattern matching for inconsistency detection into a dependency graph. Each hypervertex in the GRDG represents a detected violation in the graph instance. The set of hypervertices H is defined as the union of the results of all constraint queries:

$$H = \{\text{results}_{CQ_i} \mid \forall CQ_i \in C\}$$

Two hypervertices are connected by an undirected edge if they share any nodes in their respective subgraphs. The edge set F includes all such pairs:

$$(h_i, h_j) \in F \iff V_{h_i} \cap V_{h_j} \neq \emptyset$$

Each hypervertex overlays a subgraph corresponding to a violation, forming a layered view over the instance property graph. Violations are considered *independent* if their corresponding hypervertices do not share any node. This means changes in one can be made without affecting the other. However, shared nodes may create repair conflicts when multiple users act simultaneously. To construct the GRDG, we design an algorithm that processes the results of constraint queries. It first creates a hypervertex for each detected violation. To detect overlapping violations, a graph query identifies shared nodes across subgraphs. This avoids nested loops and improves efficiency. Edges are then added between hypervertices

with overlapping nodes. The resulting GRDG helps in managing and visualizing dependencies between inconsistencies.

4.3 User-Centric Property Graph Repairs

The GDRG is a flexible data structure that can guide the application of constraints. In our work, we consider a collaborative approach in which multiple users are requested to fix the violations [63, 64]. The process starts with the users being assigned to independent violations and continues after they choose their preferred repair. The CDRG also facilitates the verification of repair properties, such as *safety*, to check whether the repair introduces new violations. If the repair is not safe, the repair is rejected and the user needs to choose another one.

Then, before proceeding to a new repair iteration, the effect of the repair is propagated to the GDRG. First, it checks if additional violations are solved or, on the contrary—if the framework allows unsafe repair—if new violations are introduced. To avoid conflicts due to the propagation of repairs, we need to assign independent hypervertices to different users. To do so, we assign violations to users, only if the sets of neighbors of the respective hypernodes on the GRDG are disjoint.

This problem is similar to the independent set problem [83], where the goal is to find a set of vertices in a graph, none of which are adjacent. However, in our context, we need to extend this concept to account for the neighbors as well. Specifically, we need to ensure that not only are the selected hypervertices independent, but also that their neighbors do not overlap with any other selected hypervertices or their respective neighbors.

The greedy approach instead, although not guaranteeing an optimal solution, offers a practical trade-off by significantly reducing the complexity for each assignment iteration and providing a fast response to users. For this reason, we choose the Greedy Algorithm approach [45], extending it to account for independent neighbors. The greedy algorithm recursively selects the next node to add to the independent set based on the nodes' degrees, starting with the one of the lowest degree.

Since graph violations are interdependent and graph data might be large, it is important to ensure accuracy, efficiency and scalability of the underlying system. We evaluate our approach on real-world property graph data with both injected and real-world inconsistencies. We consider user simulations with different user expertise (e.g, from oracles to random users and their variants thereof) against both interactive and non interactive baselines. We have also performed a user study to assess how humans perform in the graph repairs. The results show that our approach outperforms the baselines for different sets of constraints by 30% on average repair quality even without oracle users. Moreover, our algorithms gracefully scale with respect to dataset size and number of users.

5 PROPERTY GRAPH TRANSFORMATIONS FOR CAUSAL ANALYSIS

Property graph transformations are not only instrumental for data management tasks, such as data integration and data cleaning, but they can serve as data model paradigms for Artificial Intelligence tasks, such as causal analysis and inference. Currently, these two paradigms, namely causality and graph data management, are developed by entirely separate communities with different motivations: one is interested in causal analysis and inference, realized by means of ad-hoc programs and scripts on empirically validated causal graphs, while the other is interested in collecting, validating, integrating and querying graph data by means of declarative languages. These two areas are investigated in isolation although combining them creates exciting possibilities. In our work, we have developed a vision behind the interplay of these two areas [65].

One of the most established theory of causality is represented by structural causal models (SCMs). SCMs consist of a causal graph and structural equations. Formally, a causal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph where $\mathcal{V}$ is the node set and $\mathcal{E}$ is the edge set. Each node in $\mathcal{V}$ represents a random variable while an edge $x \rightarrow y$ represents a causal effect between two variables $x, y \in \mathcal{V}$. Given an edge $x \rightarrow y$ we call x the *exposure* and y the outcome. The node set $\mathcal{V}$ contains also all the observed and unobserved variables. In particular, we consider causal directed acyclic graphs (DAGs) introduced in [33].

5.1 Past Approaches

Domain experts are oftentimes compelled to figure out the causal relationships before actually collecting any observational data. The latter data is sampled from a population of homogeneous units, thus boiling down to the unit homogeneity assumption [49]. Causal graphs are manually designed and scattered around notebooks created on an ad-hoc basis with *external libraries*, such as CDT [53], Py-Causal and Do-Why [20], and never stored, tracked and versioned in a database.

However, observational data is inherently complex, large and hides several dimensions, that are scattered around multiple heterogeneous units and that have semantic links among units (e.g. variables about a patient concerning diseases, comorbidities, diet, age, physical and mental conditions are scattered around multiple heterogeneous units). This observation makes knowledge graphs and, in particular, expressive property graphs [12, 27, 81] a perfectly suitable data model for the underlying observational data in order to be able to extract and mine candidate causal graphs. Property graph pattern matching techniques and queries represent a breakthrough for efficient causal property graph representation and extraction in graph databases. Indeed, existing libraries in Python, such as PyWhy-Graphs [44] integrated with NetworkX API, present several limitations as they only implement subsets of causal graphs or focus on edge directions as the main feature [31, 67, 86].

Early work focusing on a relational database framework [58, 71] studied how multiple relational units are collapsed in one unit using embeddings and aggregation, thus focusing on simple boolean queries, involving one causal effect at a time. The computational complexity of causality computation for relational queries is tractable, and, precisely, doable in polynomial time [59], contrarily to the general causality problem, which is set to be NP-complete [35]. The relationship among actual causes (of a given relational tuple t) of query results with preferred repairs on denial constraints and preference-based semantics has been investigated in previous work mainly for query and constraint explainability [18, 19].

Causal inference leads to estimations of causes and counterfactuals, sensitivity analysis from cause-effects and learning probabilities of causation from domain-specific populations. However, these tasks lack a data-driven approach in which causal artifacts are aligned with observational data and stored, efficiently queried, updated and analyzed in the same database system.

5.2 Causal Property Graph Transformations

Figure 4(a) shows an example of a property graph transformation from a property graph to causal DAGs in Figure 4(b). It models a person named Ali who smokes and drinks some alcohol and has two diseases (`COPD` and `Stress`), and a person named Kate suffering from the same `COPD` condition, but having also a job as an engineer.

To perform causal analysis, we need to extract causal variables from the PG and the relationships between them. Moreover, in order to apply interventions and perform causal analysis, we need to maintain a mapping between the PG instances and the causal variables in the causal DAG. It is seldom the case that one variable in a causal DAG is mapped to a single vertex in a property graph. In several cases, this mapping is not only one-to-many but also compositional. Hence, it can be encoded as a property graph transformation, as presented in Section 2.

Representing causal DAGs as property graphs offers several key advantages. First, it enables the causal DAG and the associated observational data to coexist within a unified data artifact, allowing them to be queried and analyzed jointly. Second, this representation makes it possible to leverage well-established techniques from graph data management—such as PG-Schema [9], PG-keys [10], and graph views [24, 46]—as well as powerful declarative query languages like Cypher, GQL, or SQL/PGQ for expressive and efficient analysis. We argue that causal analysis is fundamentally navigational, as causal relationships are naturally encoded as edges in a graph. The property graph model is therefore particularly well-suited since graph query languages are designed precisely to support such navigational reasoning. From the above definition, it is clear that hypernodes need to be extracted from the underlying property graphs. An example of declarative property graph transformation merging *Smoking* and *COPD* in a causal DAG is the following Cypher query, where a new relationship is generated when a condition is met (the matched pattern):

```
MATCH (p:Person)-[:HAS_HABIT]->(h:Habit),
(p)-[:HAS_CONDITION]->(c:Condition)
WHERE c.name = "COPD" AND h.type="Cigarettes" WITH h,c
MERGE (h)-[:BELONGS]->(x:SMOKING)-->(y:COPD)<-[:BELONGS]-(c)
```

However, this query produces a node (variable) for each matched path. In Figure 4(b), for example, a variable (`SMOKING`) is extracted twice from the property graph instance. To deal with duplicates, we should merge the generated nodes that have the same label. This is possible in concrete graph query languages but requires multiple complex queries. The first step of our vision consists of extending the GQL syntax with a new operator `EXTRACT`, that allows us to easily express the causal variable extraction by abstracting out the details of hypernodes and graph transformations:

```
MATCH (p:Person)-[:HAS_HABIT]->(h:Habit),
(p)-[:HAS_CONDITION]->(c:Condition)
WHERE c.name = "COPD" AND h.type="Cigarettes"
EXTRACT (x:SMOKING)-->(y:COPD)
```

After extracting the variables from the property graph instance, the causal DAG is not yet complete as it needs to be enriched with the probability distributions to the edges as properties.

Figure 4(b) shows an example of causal DAG extracted from a property graph (the colors indicate the mapping between the observational data in the property graph and the causal variables). We can see how different paths in the property graph are mapped to the causal DAG, specifically the query listed above created the path `(SMOKING)-->(COPD)` from the path including `Person` and `Habit`.

Figure 4(c) shows potential results of graph queries computing key components of causal path analysis: i) `(INCOME LEVEL)-->(STRESS)-->(SMOKING)` is a causal path representing an indirect effect of `INCOME LEVEL` on `SMOKING` through `STRESS` as *mediator* variable. ii) The path `(SMOKING) <--(AGE)-->(COPD)` is a confounding path with `AGE` as a *confounder* (e.g. having effect on both `SMOKING` and `COPD`). iii) The path `(STRESS)-->(COPD) <--(SMOKING)` is a collider path with `COPD` as *collider*.

6 CONCLUSION AND OPEN CHALLENGES

Our work focuses on property graph transformations as a novel, versatile and practical paradigm for unifying data across multiple formats and yielding a conflict-free and consistent graph view on top of these formats. We have shown the utility and effectiveness of this paradigm in data integration and data cleaning. The latter areas have achieved a great level of maturity on relational and XML data models but have been less studied for expressive property graphs. Declarative graph transformations leveraging property values for identity creation and generating complex structures, such as nodes, edges and whole subgraphs, can be expressed in current standard graph query languages with suitable extensions. Similarly, graph repairs fixing graph denial constraints for property graphs can be formalized as graph transformations. Strategies to repair graph data glitches using deletions and label/property value updates use an hypergraph overlay in order to prioritize repairs based on their respective overlap. We also showed that property graph transformations are compatible with causal analysis artifacts and, in particular, with causal DAGs from the theory of causation. Mappings between the underlying observational data and causal DAGs can be elegantly encoded as unique property graph transformations and used for a variety of causal analysis tasks, such as mediator analysis, interventions and counterfactuals analysis.

This work paves the way to several future directions of investigations, that we discuss in the remainder of this section.

Optimization of Property Graph Transformations. While we have shown efficient and executable property graph transformations on top of graph query engines, several problems on optimizing these transformations remain to be studied, especially with transformations manipulating paths and subgraphs [11]. Concerning property graph transformations, issues such as better scalability dealing with large-scale data are still open along with the design of benchmarks for property graph transformations, encompassing real-world use cases. While we have already shown property graph transformation at work with relational data, defining these operations for heterogeneous temporal data, unifying property graphs and time series and studying the algorithmic aspect of their evaluation are also interesting future endeavors.

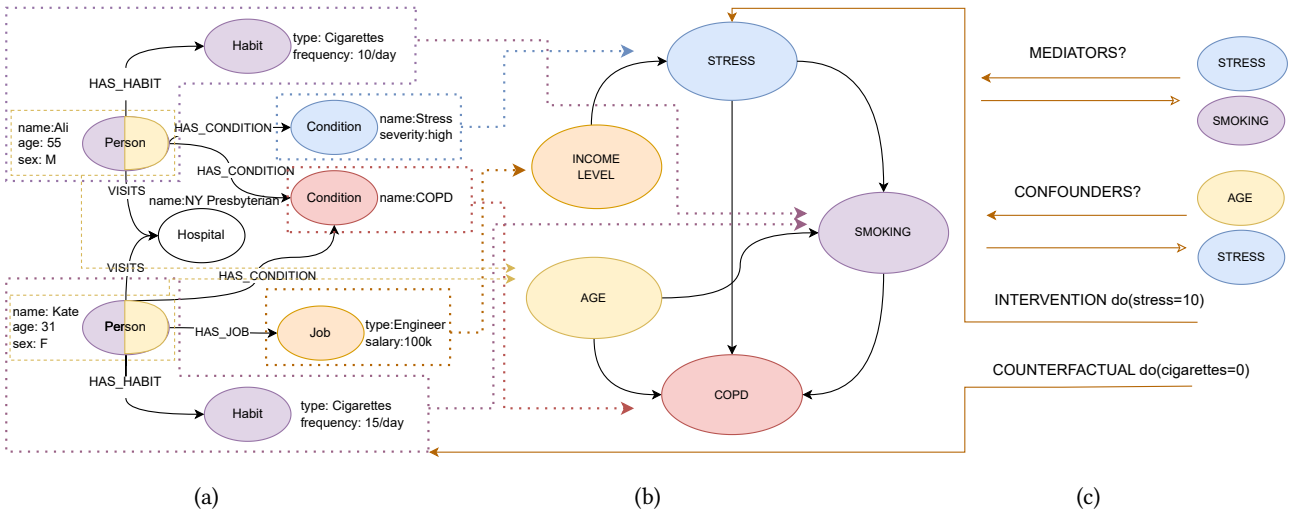


Figure 4: Example of a Causal Directed Acyclic Hypergraph model including the observational data (a), their mapping to the causal variables in the causal DAG (b), and the results of different causal queries (c).

Property graph cleaning. Data cleaning for property graphs is also a broad research area, which deserves much attention. While we have studied interactive algorithms involving several users, future work should be devoted to combine automated approaches and user-centric approaches in order to decrease the user's burden. Automated approaches for property graphs can leverage AI-centric methods, such as Graph Neural Networks (GNN), Large Language Models (LLMs) and Graph RAG (Retrieval Augmented Generation) to help classify the graph-based data glitches [66, 78]. Handling complex constraints, encompassing recursive property graph rules, is also an important challenge, since these rules span parts of the property graph that needs to be inspected at runtime due to paths of arbitrary length possibly with cycles. The efficiency of all these methods and their scalability should also be at the core of our investigation.

Unifying Graph Databases and Causal Analysis. Last but not least, several causal graph analysis problems remain to be studied by means of graph transformations. The evolution of causal DAGs mapped to the underlying observational data and relating to data fusion and transportability of causal DAGs [15] in the causality community are certainly worth investigating. They model data collection conditions, but disregard the other steps of data management, such as data integration, consistency and evolution, which can be encoded by means of metadata naturally exposed by property graphs. Notions of validity and consistency of structural equation models (SEMs) [69] in order to characterize exact transformations between causal models need to be redefined from a data management perspective and using quality-driven transformations of property graphs [24]. Last but not least, d-separation is a criterion used in causal analysis for deciding, from a given a causal graph, whether a set X of variables is independent of another set Y, given a third set of variables Z. As such, d-separation is a pure graph-theoretic notion that associates dependence of causal variables with connectivity in a graph. Hence, d-separated sets in causal DAGs can be interpreted as a series of powerful property graph transformation operations, creating property graph views to encode d-separation.

ACKNOWLEDGMENTS

I am grateful to Stefano Ceri and Chao Zhang for their insightful comments and suggestions on an early draft of this paper. I would like to thank all my co-authors and PhD students for this line of work, and, in particular, Andrea Mauri, Filip Murlak, Amedeo Pachera, Mattia Palmiotto and Yann Ramusat. Last but not least, I am thankful to Renee Miller for her 2018 Award paper, whose structure and organization inspired mine. This work was supported by the French National Research Agency (ANR-21-CE48-0015 VeriGraph) and by AAP Etoiles 2023 Lyon 1.

REFERENCES

[1] 2024. Amazon Neptune. https://aws.amazon.com/neptune/ Accessed: 2024-12-04.

[2] 2024. Kuzu Graph Database. https://kuzudb.com/ Accessed: 2024-12-04.

[3] 2024. Neo4j Graph Database Platform. https://neo4j.com/ Accessed: 2024-12-04.

[4] 2024. Oracle PGX: Parallel Graph AnalytiX. https://www.oracle.com/database/technologies/graph-analytics/ Accessed: 2024-12-04.

[5] 2024. RedisGraph Module. https://redis.io/docs/stack/graph/ Accessed: 2024-12-04.

[6] 2024. SAP HANA Graph. https://help.sap.com/viewer/product/SAP_HANA_GRAPH/ Accessed: 2024-12-04.

[7] 2024. Sparksee Graph Database. https://sparsity-technologies.com/sparksee/ Accessed: 2024-12-04.

[8] 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. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1421–1432. https://doi.org/10.1145/3183713.3190654

[9] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 198:1–198:25. https://doi.org/10.1145/3589778

[10] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2423–2436. https://doi.org/10.1145/3448016.3457561

[11] Renzo Angles, Angela Bonifati, Roberto García, and Domagoj Vrgoc. 2025. Path-based Algebraic Foundations of Graph Query Languages. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*. 783–795. https://doi.org/10.48786/EDBT.2025.63

[12] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39.

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

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

[15] Elias Bareinboim and Judea Pearl. 2016. Causal inference and the data-fusion problem. *Proc. Natl. Acad. Sci. USA* 113, 27 (2016), 7345–7352.

[16] Zohra Bellahsene, Angela Bonifati, and Erhard Rahm (Eds.). 2011. *Schema Matching and Mapping*. Springer.

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

[18] Leopoldo E. Bertossi and Babak Salimi. 2017. Causes for query answers from databases: Datalog abduction, view-updates, and integrity constraints. *Int. J. Approx. Reason.* 90 (2017), 226–252.

[19] Leopoldo E. Bertossi and Babak Salimi. 2017. From Causes for Database Queries to Repairs and Model-Based Diagnosis and Back. *Theory Comput. Syst.* 61, 1 (2017), 191–232.

[20] Patrick Blöbaum, Peter Götz, Kailash Budhathoki, Atalanti A. Mastakouri, and Dominik Janzing. 2022. DoWhy-GCM: An extension of DoWhy for causal inference in graphical causal models. *arXiv preprint arXiv:2206.06821* (2022).

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

[22] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[23] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings*. 448–456.

[24] Angela Bonifati, Filip Murlak, and Yann Ramusat. 2024. Transforming Property Graphs. *Proc. VLDB Endow.* 17, 11 (2024), 2906–2918. https://www.vldb.org/pvldb/vol17/p2906-ramusat.pdf

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

[26] Angela Bonifati, Yann Ramusat, Filip Murlak, Amela Fejza, and Rachid Echahed. 2024. DTGraph: Declarative Transformations of Property Graphs. *Proc. VLDB Endow.* 17, 12 (2024), 4265–4268.

[27] Vinay K. Chaudhri, Chaitanya K. Baru, Naren Chittar, Xin Luna Dong, Michael R. Genesereth, James A. Hendler, Aditya Kalyanpur, Douglas B. Lenat, Juan Sequeda, Denny Vrandecic, and Kuansan Wang. 2022. Knowledge Graphs: Introduction, History and, Perspectives. *AI Mag.* 43, 1 (2022), 17–29.

[28] Yurong Cheng, Lei Chen, Ye Yuan, and Guoren Wang. 2018. Rule-Based Graph Repairing: Semantic and Efficient Repairing Methods. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 773–784. https://doi.org/10.1109/ICDE.2018.00075

[29] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Information and Computation* 197, 1-2 (2005), 90–121.

[30] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 458–469.

[31] Diego Colombo and Marloes H. Maathuis. 2014. Order-independent constraint-based causal structure learning. *J. Mach. Learn. Res.* 15, 1 (2014), 3741–3782.

[32] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 251–264.

[33] Rina Dechter and Judea Pearl. 1991. Directed Constraint Networks: A Relational Framework for Causal Modeling. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*. 1164–1170.

[34] 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*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258. https://doi.org/10.1145/3514221.3526057

[35] Thomas Eiter and Thomas Lukasiewicz. 2004. Complexity results for explanations in the structural-model approach. *Artificial Intelligence* 154, 1 (2004), 145–198.

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

[37] Wenfei Fan and Ping Lu. 2017. Dependencies for Graphs. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 403–416. https://doi.org/10.1145/3034786.3056114

[38] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional Dependencies for Graphs. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1843–1857. https://doi.org/10.1145/2882903.2915232

[39] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2022. GPC: A Pattern Calculus for Property Graphs. *CoRR* abs/2210.16580 (2022).

[40] 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.

[41] 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*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM. https://doi.org/10.1145/3183713.3190657

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

[43] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2024. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *CoRR* abs/2409.01102 (2024). https://doi.org/10.48550/arXiv.2409.01102

[44] Py-Why Graphs. 2024. Py-Why-Graphs. https://github.com/py-why/pywhy-graphs. [Online; accessed 17-August-2024].

[45] Magnús Halldórsson and Jaikumar Radhakrishnan. 1994. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. 439–448.

[46] Soonbo Han and Zachary G. Ives. 2024. Implementation Strategies for Views over Property Graphs. *Proc. ACM Manag. Data* 2, 3, Article 146 (May 2024), 26 pages. https://doi.org/10.1145/3654949

[47] Soonbo Han and Zachary G. Ives. 2025. Implementing Views for Property Graphs. *SIGMOD Rec.* 54, 1 (2025), 59–68.

[48] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. 2016. Interactive and Deterministic Data Cleaning. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 893–907. https://doi.org/10.1145/2882903.2915242

[49] Paul W. Holland. 1986. Statistics and causal inference. *J. Amer. Statist. Assoc.* 81, 396 (1986), 945–960.

[50] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. ACM Books, Vol. 28. ACM. https://doi.org/10.1145/3310205

[51] ISO. 2024. ISO/IEC 39075:2024 Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html.

[52] Paul Juillard, Angela Bonifati, and Andrea Mauri. 2024. Interactive Graph Repairs for Neighborhood Constraints. In *Proceedings 27th International Conference on Extending Database Technology (EDBT 2024) Paestum, Italy, March 25 - March 28*. OpenProceedings.org, 2:175–2:187.

[53] Diviyan Kalainathan, Olivier Goudet, and Ritik Dutta. 2020. Causal Discovery Toolbox: Uncovering causal relationships in Python. *J. Mach. Learn. Res.* 21 (2020), 37:1–37:5.

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

[55] Haridimos Kondylakis, Stefania Dumbrava, Matteo Lissandrini, Nokolay Yakovets, Angela Bonifati, Vasilis Efthymiou, George Fletcher, Dimitris Plexousakis, Riccardo Tommasini, Georgia Troullinou, and Elisjana Ymeralli. 2025. Property Graph Standards: State of the Art & Open Challenges. *Proc. VLDB Endow.* 18, 12 (2025), 5477–5481.

[56] Maurizio Lenzerini. 2002. Data integration: a theoretical perspective. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Madison, Wisconsin) *(PODS '02)*. Association for Computing Machinery, New York, NY, USA, 233–246. https://doi.org/10.1145/543613.543644

[57] Peng Lin, Qi Song, Yinghui Wu, and Jiaxing Pi. 2020. Repairing entities using star constraints in multirelational graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 229–240.

[58] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y. Halpern, Christoph Koch, Katherine F. Moore, and Dan Suciu. 2010. Causality in Databases. *IEEE Data Eng. Bull.* 33, 3 (2010), 59–67. http://sites.computer.org/debull/A10sept/suciu.pdf

[59] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *Proc. VLDB Endow.* 4, 1 (2010), 34–45. https://doi.org/10.14778/1880172.1880176

[60] Amine Mhedhbi, Amol Deshpande, and Semih Salihoglu. 2024. Modern Techniques For Querying Graph-structured Databases. *Found. Trends Databases* 14, 2 (2024), 72–185. https://doi.org/10.1561/1900000090

[61] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. 2000. Schema Mapping as Query Discovery. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. 77–88.

[62] Inc. Neo4j. [n.d.]. ICIJ offshore leaks. Retrieved October 2, 2024 from https://github.com/neo4j-graph-examples/icij-offshoreleaks

[63] Amedeo Pachera, Angela Bonifati, and Andrea Mauri. 2025. Grafixer: Enabling User-Centric Repairs for Property Graphs. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. 199–202.

[64] Amedeo Pachera, Angela Bonifati, and Andrea Mauri. 2025. User-Centric Property Graph Repairs. *Proc. ACM Manag. Data* 3, 1 (2025), 85:1–85:27.

[65] Amedeo Pachera, Mattia Palmiotto, Angela Bonifati, and Andrea Mauri. 2025. What If: Causal Analysis with Graph Databases. *Proc. VLDB Endow.* 18, 11 (2025), 4009–4016.

[66] Jeff Z. Pan, Simon Razniewski, Jan-Christoph Kalo, Sneha Singhania, Jiaoyan Chen, Stefan Dietze, Hajira Jabeen, Janna Omeliyanenko, Wen Zhang, Matteo Lissandrini, Russa Biswas, Gerard de Melo, Angela Bonifati, Edlira Vakaj, Mauro Dragoni, and Damien Graux. 2023. Large Language Models and Knowledge Graphs: Opportunities and Challenges. *TGDK* 1, 1 (2023), 2:1–2:38.

[67] Jonas Peters, Dominik Janzing, Arthur Gretton, and Bernhard Schölkopf. 2009. Detecting the direction of causal time series. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009 (ACM International Conference Proceeding Series)*, Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman (Eds.), Vol. 382. ACM, 801–808.

[68] Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. 2025. Transforming RDF Graphs to Property Graphs using Standardized Schemas (To appear). *Proc. ACM Manag. Data* 3 (2025), 198:1–198:25.

[69] Paul K. Rubenstein, Sebastian Weichwald, Stephan Bongers, Joris M. Mooij, Dominik Janzing, Moritz Grosse-Wentrup, and Bernhard Schölkopf. 2017. Causal Consistency of Structural Equation Models. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*.

[70] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The Future is Big Graphs: A Community View on Graph Processing Systems. *Commun. ACM* 64, 9 (aug 2021), 62–71. https://doi.org/10.1145/3434642

[71] Babak Salimi, Harsh Parikh, Moe Kayali, Lise Getoor, Sudeepa Roy, and Dan Suciu. 2020. Causal Relational Learning. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 241–256.

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

[73] 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

[74] Rajesh Shrestha, Omeed Habibelahian, Arash Termehchy, and Paolo Papotti. 2023. Exploratory Training: When Annonators Learn About Data. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[75] Shaoxu Song, Boge Liu, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. 2017. Graph repairing under neighborhood constraints. *The VLDB Journal* 26, 5 (2017), 611–635.

[76] Slawek Staworko, Jan Chomicki, and Jerzy Marcinkowski. 2012. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.* 64, 2-3 (2012), 209–246. https://doi.org/10.1007/s10472-012-9288-8

[77] Michael Stonebraker and Ihab F. Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.

[78] Hrishikesh Terdalkar, Angela Bonifati, and Andrea Mauri. 2025. Graph Repairs with Large Language Models: An Empirical Study. In *Proceedings of the 8th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Berlin, Germany, June 22-27, 2025*. 9:1–9:10.

[79] Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, and Nan Tang. 2017. UGuide: User-Guided Discovery of FD-Detectable Errors. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1385–1397. https://doi.org/10.1145/3035918.3064024

[80] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Vicente Calisto, Benjamín Farias, Sebastián Ferrada, Tristan Heuer, Aidan Hogan, Gonzalo Navarro, Alexander Pinto, Juan L. Reutter, Henry Rosales-Méndez, and Etienne Toussaint. 2024. MillenniumDB: A Multi-modal, Multi-model Graph Database. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*. 496–499.

[81] Gerhard Weikum. 2021. Knowledge Graphs 2021: A Data Odyssey. *Proc. VLDB Endow.* 14, 12 (2021), 3233–3238.

[82] Jef Wijsen. 2005. Database repairing using updates. *ACM Transactions on Database Systems (TODS)* 30, 3 (2005), 722–768.

[83] Mingyu Xiao and Hiroshi Nagamochi. 2017. Exact algorithms for maximum independent set. *Information and Computation* 255 (2017), 126–146.

[84] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided data repair. *Proc. VLDB Endow.* 4, 5 (feb 2011), 279–289. https://doi.org/10.14778/1952376.1952378

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

[86] Boxiang Zhao, Shuliang Wang, Lianhua Chi, Qi Li, Xiaojia Liu, and Jing Geng. 2023. Causal Discovery via Causal Star Graphs. *ACM Trans. Knowl. Discov. Data* 17, 7, Article 98 (apr 2023), 24 pages.