A Graph-Native Approach to Normalization

Johannes Schrott
johannes.schrott@tuwien.ac.at
TU Wien
Vienna, Austria

Maxime Jakubowski
maxime.jakubowski@tuwien.ac.at
TU Wien
Vienna, Austria

Katja Hose
katja.hose@tuwien.ac.at
TU Wien
Vienna, Austria

Abstract

In recent years, knowledge graphs (KGs) – in particular in the form of labeled property graphs (LPGs) – have become essential components in a broad range of applications. Although the absence of strict schemas for KGs facilitates structural issues that lead to redundancies and subsequently to inconsistencies and anomalies, the problem of KG quality has so far received only little attention. Inspired by normalization using functional dependencies for relational data, a first approach exploiting dependencies within nodes has been proposed. However, real-world KGs also expose functional dependencies involving edges. In this paper, we therefore propose graph-native normalization, which considers dependencies within nodes, edges, and their combination. We define a range of graph-native normal forms and graph object functional dependencies and propose algorithms for transforming graphs accordingly. We evaluate our contributions using a broad range of synthetic and native graph datasets.

Keywords

Knowledge Graphs, Labeled Property Graphs, Knowledge Graph Quality, Normalization, Redundancy

1 Introduction

Labeled property graphs (LPGs) have emerged as a prominent form of knowledge graphs (KGs), seeing extensive use in a variety of contexts, ranging from data integration [22] to fraud detection [23], and are a central abstraction in modern graph processing systems [32]. Especially with their recent popularity as a source of reliable knowledge in machine learning (ML) and artificial intelligence (AI), e.g., to mitigate hallucinations in large language models (LLMs) [21], ensuring the quality of KGs becomes more and more essential [19]. As a first step towards achieving this goal, new standards and schema-level approaches have been proposed, e.g., the Graph Query Language (GQL) [16, 20], PG-Schema [3], PG-Keys [4], and transformations or alignments between them [2, 31]. However, being able to define schemas is only the first step. We still need to fully understand what a good schema is and how we can improve the schema of a KG. In relational databases, normalization [6, 10, 11] is a well-known technique to transform a database schema into a schema that is commonly perceived as better with respect to a number of desirable properties. Such transformations are traditionally guided by exploiting functional dependencies (FDs) and are described by normal forms, which generally increase data quality by reducing redundancy; the latter not only unnecessarily inflates storage space but also serves as a source of error since all redundant information needs to be kept in sync to ensure consistency.

Originally introduced for the relational data model in the 1970s [10, 11], normalization has recently been adopted to the LPG setting by Skavantzos and Link [36, 37]. In these works, nodes are considered similar to relations that are being normalized. However, graphs are subject to additional types of redundancies. Consider, for example, the graph depicted in Figure 1a, which captures information about courses and lecturers. In this graph, we see that it might be worthwhile to enforce that *each course is taught using exactly one book*. However, this is difficult to achieve in the current graph because the information is modeled as a property of edges with label teaches that connect teachers to courses. This way of modeling the information might have historical reasons but as the graph keeps on growing and evolving, it might lead to errors. In Figure 1a, redundancy is indicated by dotted lines. Hence, to reduce the degree of redundancy and therefore the potential for inconsistencies, the graph should be normalized so that the information about the book is stored together with the course instead of the teaches edges (see Figure 2). In this paper, we build upon the work proposed by Skavantzos and Link [37], which focuses on dependencies within nodes, and extend it to support also dependencies within edges, and dependencies between nodes and edges. Since our approach focuses on both, nodes and edges, we call it *graph-native*. In summary, the contributions of this paper are as follows:

(i) We propose a new dependency type, Graph Object Functional Dependencies (GO-FDs), that expresses dependencies within and between nodes and edges.
(ii) Building upon the state of the art, we provide graph-native generalizations of existing normal forms, which we call Graph-Native Normal Forms (GN-NFs).
(iii) We identify redundancy patterns for LPGs and transformations to remove them. Based thereon, we introduce graph-native LPG normalization algorithms that transform a given graph into our GN-NFs.
(iv) Our experimental evaluation shows how graph-native normalization guided by GO-FDs reduces redundancy and affects query performance.

This paper is structured as follows: Section 2 introduces basic concepts and a running example; Section 3 reviews related work; Section 4 defines GO-FDs for LPGs; Section 5 analyzes redundancies and defines GN-NFs; Section 6 presents our normalization algorithms; Section 7 reports evaluation results; and Section 8 concludes with a summary and future work.

2 Preliminaries and Running Example

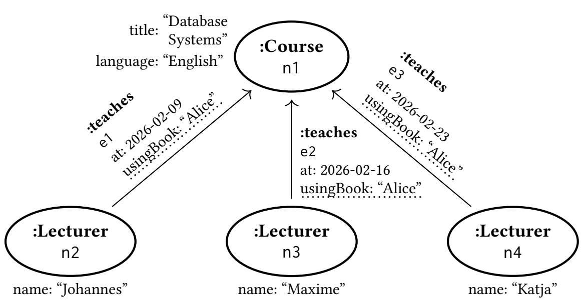
Since this paper relies on notions from the relational data model, including functional dependencies (FDs), and applies them to labeled property graphs (LPGs), we provide preliminary notions for these based on the literature [1, 3, 8, 16].

Labeled Property Graphs. We assume pairwise disjoint countable sets of labels $\mathbf{L}$, property keys $\mathbf{P}$, constants $\mathbf{C}$, and (graph) object identifiers $\mathbf{O}$. We often refer to object identifiers as *(graph) objects*: they represent either *nodes* or *edges*. A labeled property graph G is a tuple $(N, E, \mathrm{lab}, \mathrm{src}, \mathrm{tgt}, \mathrm{prop})$ with:

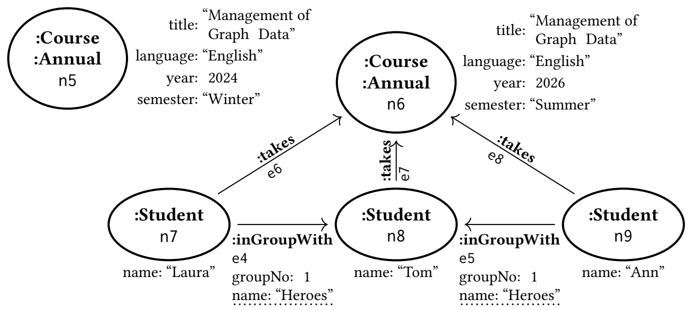
- mutually disjoint finite subsets N, E of $\mathbf{O}$ that represent the nodes and edges respectively;

EDBT '27, Lille (France)

10.48786/edbt.2027.06



(a) Courses taught by lecturers using books



(b) Courses taken by students that work together in groups

Figure 1: Examples of denormalized LPGs about courses, lecturers, and students. Redundancies are underlined with dots.

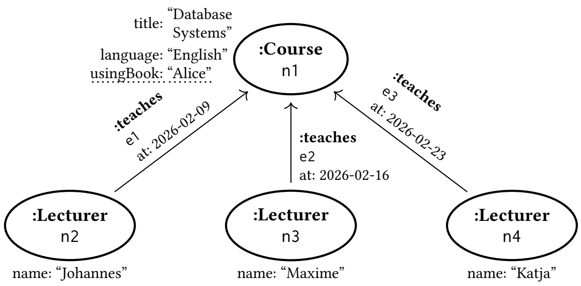


Figure 2: Normalized version of the LPG from Figure 1a

- a total function lab : $N \cup E \to 2^{\mathbf{L}}$ that maps objects to (possibly empty) sets of labels;
- total functions src : $E \to N$ and tgt : $E \to N$ that map each edge to nodes representing the edge's source and target respectively; and
- a partial function prop : $(N \cup E) \times \mathbf{P} \to \mathbf{C}$ that assigns constants to the property keys in objects.

This definition of LPGs is compatible with GQL [20] and common graph database systems, such as Neo4j [26] or Memgraph [25]. The data model of these two systems is a subset of our definition, where, contrary to GQL, edges can only have exactly one label, which is called type.

Relation Schemas, Tuples, and Relations. We assume mutually disjoint countable sets of *relation names* $\mathcal{R}$ and *attributes* $\mathcal{A}$, disjoint with any of the previous sets. When we assume a finite set of attributes $A \subset \mathcal{A}$ with a relation name $R \in \mathcal{R}$, denoted by $\mathsf{attr}(R) = A$, we call R a relation schema with attributes A. A *tuple* μ for a relation schema R is a function that assigns constants from $\mathbf{C}$ to $\mathsf{attr}(R)$. Formally, it is a total function $\mu : \mathsf{attr}(R) \mapsto \mathbf{C}$. An *instance* I of a relation schema R is a set of tuples for R. Sometimes, I is simply called a *relation*. We say that two tuples μ_1, μ_2 *agree* on attributes A if for every $a \in A$, $\mu_1(a) = \mu_2(a)$.

Functional Dependencies and Keys. Essential to all techniques related to normalization are so-called dependencies. A dependency is a type of constraint that holds on multiple data records – it can cover multiple data attributes [9]. The dependency type relevant for normalization in the relational data model are *functional dependencies (FDs)*. FDs express "which attributes are functional dependent on others" [11]. In other words, this means that the values of a certain set of attributes X define the values of another set of attributes Y. Formally, an FD over a set of attributes A is an

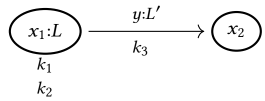


Figure 3: Example of the graphical graph pattern notation

expression $X \Rightarrow Y$, with $X, Y \subseteq A$. An FD $X \Rightarrow Y$ is *applicable* to a relation schema R if $X \cup Y \subseteq \mathsf{attr}(R)$. We say that an FD $X \Rightarrow Y$ is satisfied by a relation instance I for R, denoted by $I \models X \Rightarrow Y$, if for every two tuples $\mu_1, \mu_2 \in I$, whenever they agree on X, they agree on Y. Given a set of FDs Σ over A, and another dependency $X \Rightarrow Y$ we say that $X \Rightarrow Y$ holds in Σ, denoted by $\Sigma \models X \Rightarrow Y$, if for every instance I of a relation schema R with $\mathsf{attr}(R) = A$ that satisfies all dependencies in Σ, we have $I \models X \Rightarrow Y$. We assume for every relation schema R a (possibly empty) set of FDs $\mathsf{deps}(R)$. A *superkey* of a relation schema R is a set $X \subseteq \mathsf{attr}(R)$ such that $\mathsf{deps}(R) \models X \Rightarrow \mathsf{attr}(R)$. A *key* for R is a (subset-)minimal superkey.

Graph Patterns. In the LPG literature, the notion of graph patterns is often used to express queries. We adopt this notion for our purposes and consider a semantics that corresponds to the *graph pattern calculus (GPC)* [15]. We do not need the full power of GPC, specifically, we do not consider repetitions. Throughout this paper, we use two notations for patterns: (i) a graphical notation and (ii) a concrete syntax.

The graphical notation consists of nodes and edges. Nodes are circles, edges are arrows. Both nodes and edges have a variable (typically x or y), an associated set of labels (typically denoted by L), and a set of property keys (typically $P = \{k_1, k_2, \dots\}$). Edges are directed. Throughout the paper, we explicitly state when edges in our notation represent both directions. Figure 3 illustrates the graphical notation with an example. Here, x_1 matches nodes that have (at least) labels L and property keys k_1 and k_2. Node x_1 has an outgoing edge y with (at least) labels L' and property key k_3 to another node x_2 without any requirements on its labels. This graphical notation corresponds to the following GPC expression:[1]

$$(x_1{:}L)_{x_1.k_1=x_1.k_1 \wedge x_1.k_2=x_1.k_2} \xrightarrow{y:L'}_{y.k_3=y.k_3} (x_2)$$

Additionally, we introduce a concrete syntax for a subset of GPC, which we call *basic graph patterns* and often simply refer to as *graph patterns*. We assume a countable set of *object identity variables* $\mathcal{V}$, disjoint from $\mathbf{L} \cup \mathbf{P} \cup \mathbf{C}$. Let $x \in \mathcal{V}$. A *property*

[1] We use the filter condition $x_1.k_1 = x_1.k_1$ to ensure property k_1 is present.

Table 1: The evaluation $[\![Q]\!]^G$ of pattern Q in graph G

Q	$[\![Q]\!]^G$
$(x{:}L{:}P)$	$\left\{ \left\{\begin{array}{l} x \mapsto n; \\ x.k \mapsto k_v \\ \text{for each } k \in P \end{array}\right\} \middle\vert \begin{array}{l} n \in \mathrm{N}, \mathrm{lab}(n) \subseteq L, \text{ and} \\ \mathrm{prop}(n,k) = k_v \text{ for each } k \in P \end{array} \right\}$
$() \xleftrightarrow{x:L:P} ()$	$\left\{ \left\{\begin{array}{l} x \mapsto e; \\ x.k \mapsto k_v \\ \text{for each } k \in P \end{array}\right\} \middle\vert \begin{array}{l} e \in E, \mathrm{lab}(e) \subseteq L, \text{ and} \\ \mathrm{prop}(e,k) = k_v \\ \text{for each } k \in P \end{array} \right\}$
$(x{:}L{:}P) \xleftrightarrow{y:L':P'} ()$	$\left\{ \mu \cup \left\{\begin{array}{l} y \mapsto e; \\ y.k \mapsto k_v \\ \text{for each } k \in P' \end{array}\right\} \middle\vert \begin{array}{l} \mu \in [\![(x{:}L{:}P)]\!]^G \\ e \in E, \mathrm{src}(e) = \mu(x), \\ \mathrm{lab}(e) \subseteq L', \text{ and} \\ \mathrm{prop}(e,k) = k_v \\ \text{for each } k \in P' \end{array} \right\}$

variable is then an expression of the form $x.k$ with $k \in \mathbf{P}$. A *variable* is either a property- or an object identity variable. Based thereon, we define the syntax for basic graph patterns as follows:

$$Q ::= (x{:}L{:}P) \mid () \xleftrightarrow{y:L:P} () \mid (x{:}L{:}P) \xleftrightarrow{y:L':P'} ()$$

where $\leftrightarrow$ is either $\rightarrow$ or $\leftarrow$, $x, y \in \mathcal{V}$ are object identity variables, $L, L' \subseteq \mathbf{L}$ is a set of labels, and $P, P' \in \mathbf{P}$ is a set of keys. Given a graph pattern Q, we define its *output attributes* $\mathrm{attr}(Q)$ as the set of variables that occur in the pattern. We have:

$$\mathrm{attr}((x{:}L{:}P)) := \{x.k \mid k \in P\} \cup \{x\}$$
$$\mathrm{attr}(() \xleftrightarrow{y:L:P} ()) := \{y.k \mid k \in P\} \cup \{y\}$$
$$\mathrm{attr}((x{:}L{:}P) \xleftrightarrow{y:L':P'} ()) := \{x.k \mid k \in P\} \cup \{y.k \mid k \in P'\} \cup \{x, y\}$$

For an LPG G and a graph pattern Q, the evaluation of Q in G is the relation $[\![Q]\!]^G$ for a relation schema R_Q with $\mathrm{attr}(R_Q) = \mathrm{attr}(Q)$, as defined in Table 1.

Graph Transformations. Finally, we adapt the notion of graph transformations from Bonifati [7] for our purposes. A transformation is a pair (Q_{before}, Q_{after}) where Q_{before} is a graph pattern, and Q_{after} is a graph pattern that contains constructors. There are two kinds of constructors: node identity- and label constructors. Essentially, they are functions mapping sets of values to newly created node identities and labels respectively.[2] We will use the notation $\langle x \rangle$ and $\underline{x}$ to represent the creation of a new node identity and a new label based on the value x. Consider the first row in Table 4 (with $\mathbf{ID}$=$\psi_{\text{within-n}}$, disregarding the gray arrows for now). The second column (T) represents Q_{before} and is a single node pattern. It matches nodes x with labels L where the property keys k_1 and k_2 are present. Then, the pattern in the third column (T_{new}) is Q_{after}, which constructs a new node identity based on L and the property key k_1, and also constructs a new label based on the same values. Furthermore, the property keys (and associated values) k_1 and k_2 are moved from node x to node $\langle Lk_1 \rangle$. The semantics of a single transformation is defined by a *creation phase* in which new graph objects and values are created, followed by a *removal phase* in which graph objects and values are removed. For a set of graph transformations $\mathcal{T}$, first, for each transformation $T \in \mathcal{T}$, the creation phase is executed, and then, for each $T \in \mathcal{T}$, the removal phase is executed.

Running Example

To illustrate our concepts and contributions, we introduce a running example (cf. Figure 1). It captures an academic context, where courses are taught by lecturers using certain books. Some courses are taught once, others annually. The example also captures students who are taking courses and may form groups to solve course assignments together.

In this example, we can formulate several dependencies:

φ_{Course}: "Courses are identified by their title and year."

φ_{Book}: "All lecturers teaching a particular course must use the same textbook."

φ_{GroupNo}: "The assigned number of a student group determines its name."

φ_{Annual}: "Annual courses may only be taught in one of the two semesters of a year."

We can classify these dependencies on the graph objects they use. Dependency φ_{Course} is defined on nodes (Courses) and their property keys (title and year). As this dependency only involves node information, we call it a *within-node* dependency. Dependency φ_{Book} involves nodes (Lecturers) but also property keys from related edges (usingBook). We call φ_{Book} a *between-graph-object* dependency, as it involves both, nodes and edges. The last type of dependency involves only edges and their property keys. Dependency φ_{GroupNo} is such an example, and we call it a *within-edge* dependency. Dependency φ_{Annual} is another example of a within-node dependency.

[2] These constructors are better known as *Skolem functions* [7].

Table 2: Characteristics of relational normal forms [1, 18]

Relational normal form	Characteristic
1NF	All values in the relation are atomic.
2NF	Non-key attributes of the relation schema are fully dependent on keys.
3NF	Non-key attributes are not transitively dependent on keys.
BCNF	Every FD is trivial or contains a superkey on its left-hand side.

3 Related Work

Before continuing, we review existing work on (non-)relational normalization and (functional) dependencies for graphs.

3.1 Relational Normalization

The foundations of schema *normalization* are inseparably connected to the *relational model* [10]. The first normal form (1NF) has been proposed together with the relational model by Codd [10] in 1970, followed by the second normal form (2NF) [11], the third normal form (3NF) [11], and the Boyce-Codd Normal Form (BCNF) [12] shortly thereafter. An overview of the characteristics of each relational normal form is shown in Table 2.

The standard process of transforming a relation schema that is not in some desired normal form into a set of relation schemas is called *decomposition*. Decomposing a relation schema brings advantages such as smaller relations, independent information stored separately, and the omission of redundancies [1, 29]. For the relational model, decomposition techniques are primarily concerned with *functional dependencies* (cf. Section 2 and Table 2) to achieve 3NF and BCNF schemas. For FDs, a sound and complete set of reasoning rules, the so-called Armstrong's Axioms, is known and can be used, for example, to identify redundant dependencies [1, 5, 13].

Normalization is the process of decomposing a given relation schema R in order to reach 2NF, 3NF, and BCNF, effectively splitting R into n new relation schemas $R_1, \ldots, R_n$ based on its attributes, meaning $\mathsf{attr}(R) = \bigcup_{i=1}^{n} \mathsf{attr}(R_i)$. Algorithms that perform decomposition are the *synthesis algorithm* [1], which guarantees 3NF for the decomposed relation schemas, and the *decomposition algorithm* [1], which ensures BCNF. Until today, normalization remains a crucial task when designing (relational) schemas [18], as it omits anomalies (cf. [1]), reduces redundancies and the potential of inconsistencies, and it optimizes the performance of databases [39].

3.2 Non-Relational Normalization

Normalization can be applied beyond the relational model. Thalheim [39] introduced *normalization for the entity-relationship (ER) model*, led by the same objectives as relational normalization: operation anomalies, inconsistent data, and redundancies should be avoided. Unlike relational normalization, which considers only single relations (*local normalization*), ER normalization can consider whole diagrams or parts of diagrams (*global normalization*).

ER normalization considers the normalization problem, where the goal is to determine whether a transformation (called "infomorphism" in [39, 40]) ψ exists that can be applied to an ER diagram E such that it fulfills the properties of a particular normal form (cf. [39] for details on the properties). One important property is that these transformations need to be *reversible*, i.e., $\psi_{\text{inv}}(\psi(E)) = E$ holds for ψ and its inverse ψ_{inv}.

Normalization for LPGs was first proposed by Skavantzos and Link [36, 37]. LPGs are normalized using a decomposition procedure. In simplified terms, the principle is to first transform a set of nodes with properties P and labels L into a relation with attributes P. Then, this relation is decomposed, and, finally, the obtained relations are transformed back into an LPG. This approach therefore reduces redundancy by resolving within-node dependencies but does not consider the additional types of dependencies that can occur within edges or graph patterns (i.e., between-graph-object dependencies). Hence, in this paper, we propose a generalized approach to close this gap by considering these types of dependencies as well.

3.3 Functional Dependencies for Graphs

In the context of LPG normalization, Skavantzos et al. [36–38] proposed *Graph-tailored Functional Dependencies (gFDs)* to express FDs within nodes, and *Graph-tailored Uniqueness Constraints (gUCs)* to express keys. gFDs and gUCs are defined on a subset of properties of a subset of nodes N that have assigned a set of labels $L \subseteq \mathbf{L}$ and have defined values for property keys $P \subseteq \mathbf{P}$. The semantics of gFDs is similar to the semantics of FDs. For example, the gFD

$$\varphi_{\text{gFD-Annual}} := \{\mathsf{Course}, \mathsf{Annual}\} : \{\mathsf{title}, \mathsf{year}, \mathsf{semester}\} : \{\mathsf{title}, \mathsf{year}\} \Rightarrow \{\mathsf{semester}\}$$

expresses dependency φ_{Annual} (cf. Section 2). Similarly, a gUC

$$\varphi_{\text{gUC-Course}} := \{\mathsf{Course}\} : \{\mathsf{title}, \mathsf{year}\} : \{\mathsf{title}, \mathsf{year}\}$$

expresses dependency φ_{Course}, i.e., courses are identified by their title and year. Notably, gFDs and gUCs are concerned with nodes. In our terminology, they can only express within-node dependencies.

Besides gFDs and gUC, a variety of other dependencies applicable to LPGs has been defined. Including gFDs, Manouvrier and Belhajjame [24] analyzed 13 types of dependencies. However, the focus of most of these dependency types is on the identification of graph objects, on the specification of integrity constraints involving constant values, or on inference of graph objects. Except for the previously mentioned gFDs, none of the dependency types was designed for the purpose of *normalizing* graphs [24].

Hellings et al. [17] defined Functional Constraints (FCs), not for LPGs, but for arbitrary n-ary relations. They define an axiomatic system to allow reasoning over FCs. In particular, FCs can be used to define FDs over RDF-based graphs.

Since Section 6 defines a *graph-native* approach to normalization based on dependencies involving nodes and edges, we need a dependency type tailored to this task.

4 Graph Object Functional Dependencies

As a first step towards graph-native normalization, we define Graph Object Functional Dependencies (GO-FDs) as a dependency type for LPGs that goes beyond within-node dependencies. A GO-FD φ is an expression of the form:

$$\varphi ::= Q :: X \Rightarrow Y$$

where Q is a graph pattern and X, Y are sets of variables (i.e., object identity- or property variables; cf. Section 2). We refer to the pattern Q of a GO-FD as its *scope*, and to $X \Rightarrow Y$ as its *descriptor* D. We call a GO-FD $Q :: X \Rightarrow Y$ *trivial* if $Y \subseteq X$.

Whether a dependency is a *within*- or *between*-graph-object dependency simply depends on the number of graph object identity variables involved in the dependency. We make this more precise:

Definition 4.1. We call a GO-FD $\varphi = Q :: X \Rightarrow Y$ a *within*-graph-object dependency when the variables in $X \cup Y$ use exactly one object identity variable x. If x occurs as a node (resp. edge) in the scope Q, we call it a *within-node* (resp. edge) dependency. Otherwise, when two object identity variables occur in $X \cup Y$, we call φ a *between*-graph-object dependency.

Recall that graph patterns Q match subgraphs in a given graph G, with the requirement that certain sets of labels L and keys P must be present in the matched graph objects. The result of evaluating a graph pattern in a given graph is a relation where the relation schema consists of attributes that are given by the variables mentioned in the pattern. Then, FD $X \Rightarrow Y$ must hold in this relation.

Formally, a graph G satisfies a GO-FD $Q :: X \Rightarrow Y$ if $[\![Q]\!]^G \models X \Rightarrow Y$. Furthermore, we call a set of GO-FDs Σ a GN-Schema. Therefore, more generally, we say a graph G satisfies a schema Σ, denoted by $G \models \Sigma$ if it satisfies each of the dependencies in Σ.

Example 4.2. By using GO-FDs, we are able to express all textually described dependencies of the running example (cf. Section 2).

$$\varphi_{\text{Course}}\colon\ (x{:}\{\mathsf{Course}\}{:}\{\mathsf{title}, \mathsf{year}\}) :: x.\mathsf{title}, x.\mathsf{year} \Rightarrow x$$

$$\varphi_{\text{Book}}\colon\ () \xrightarrow{y:\{\mathsf{Teaches}\}:\{\mathsf{using}\}} (x{:}\{\mathsf{Course}\}{:}\emptyset) :: x \Rightarrow y.\mathsf{using}$$

$$\varphi_{\text{GroupNo}}\colon\ () \xrightarrow{y:\{\mathsf{inGroupWith}\}:\{\mathsf{groupNo},\mathsf{name}\}} () :: y.\mathsf{groupNo} \Rightarrow y.\mathsf{name}$$

$$\varphi_{\text{Annual}}\colon\ (x{:}\{\mathsf{Course}, \mathsf{Annual}\}{:}\{\mathsf{title}, \mathsf{year}, \mathsf{sem}\}) :: x.\mathsf{title}, x.\mathsf{year} \Rightarrow x.\mathsf{sem}$$

Dependency φ_{Course} acts as a uniqueness (superkey) constraint and shows how to express gUCs using GO-FDs. Dependency φ_{Book} highlights our generalization of gFDs; we are able to express dependencies *between* graph objects. Similarly, φ_{GroupNo} shows how GO-FDs can express within-edge dependencies. Finally, example φ_{Annual} shows how to express gFDs using within-node GO-FDs.

Table 3: Conditions for $Q \sqsupseteq Q'$

Q	Q'	$Q \sqsupseteq Q'$ if
$(x{:}L{:}P)$	$(x'{:}L'{:}P')$	$L' \subseteq L$ and $P' \subseteq P$
$(x{:}L{:}P)$	$(x'{:}L'{:}P') \xleftrightarrow{y:L'':P''} ()$	$L' \subseteq L$ and $P' \subseteq P$
$(x{:}L_x{:}P_x) \xleftrightarrow{y:L_y:P_y} ()$	$(x'{:}L'_x{:}P'_x) \xleftrightarrow{y:L'_y:P'_y} ()$	$L'_x \subseteq L_x$, $P'_x \subseteq P_x$; $L'_y \subseteq L_y$, $P'_y \subseteq P_y$

Axiomatization. Functional dependencies in the relational model have an axiomatization called *Armstrong's axioms.* Analogously, we also have reasoning rules for GO-FDs. First, we state axioms that are intrinsic to LPGs. We call the following three axioms *structurally implied* because they are given by the graph structure of the data model. Let G be a graph.

Object Identity (Node) $G \models (o{:}\emptyset{:}k) :: o \Rightarrow o.k$
Object Identity (Edge) $G \models () \xrightarrow{o:\emptyset:k} () :: o \Rightarrow o.k$
Edge Endpoint $G \models (x) \xrightarrow{e} (y) :: e \Rightarrow x$ and $e \Rightarrow y$

Before we continue with the associated reasoning rules, we need to introduce the notion of *pattern inclusion.* Let Q, Q' be basic graph patterns. We say Q is more general than Q', which we write as $Q \sqsupseteq Q'$ if, for any graph G, $[\![Q]\!]^G \supseteq \pi_{\mathrm{attr}(Q)}([\![Q']\!]^G)$.[3] We characterize basic graph pattern inclusion as follows:

Lemma 4.3. *$Q \sqsupseteq Q'$ iff they are as shown in Table 3.*

Proof sketch. One can verify that the relations shown in Table 3 hold. Furthermore, the remaining case is where an edge occurs in Q, but not in Q'. Suppose $Q := (x{:}L{:}P) \xrightarrow{y:L':P'} ()$ and $Q' := (x'{:}L''{:}P'')$. Then, even if $L'' \subseteq L$ and $P'' \subseteq P$, pattern Q is more restrictive since it requires the presence of an edge. □

Furthermore, Table 3 gives us a straightforward algorithm to check whether $Q' \sqsubseteq Q$ by matching the patterns and checking the subsets of property keys and labels.

We now state the reasoning rules for GO-FDs. Let Q, Q' be basic graph patterns.

Reflexivity If $Y \subseteq X$, then $Q :: X \Rightarrow Y$.
Augmentation If $Q :: X \Rightarrow Y$ and $Z \subseteq \mathrm{attr}(Q)$, then $Q :: XZ \Rightarrow YZ$.
Decomposition If $Q :: X \Rightarrow ZY$, then $Q :: X \Rightarrow Y$.
Restriction If $Q :: X \Rightarrow Y$, and $Q' \sqsubseteq Q$, then $Q' :: X \Rightarrow Y$.
Transitivity If $Q :: X \Rightarrow Y$, $Q' :: Y \Rightarrow Z$ and $Q \sqsubseteq Q'$, then $Q' :: X \Rightarrow Z$.

Theorem 4.4. *The axioms above provide a sound and complete axiomatization of GO-FDs.*

proof sketch. In case $Q = Q'$ the soundness and completeness of the axiomatization, within the scope Q, follows from Armstrong's Axioms. However, generally, the completeness of the axiomatization is non-trivial. To show completeness, we rely on the insight that GO-FDs can be encoded into *Functional Constraints (FCs)* over n-ary relations, as defined by Hellings et al. [17]. In their work, they provide a complete axiomatization of FCs, which we can use to show the completeness of our axiomatization. To this end, we must verify that (1) we can encode LPGs in a n-ary relation; (2) we can translate GO-FDs into FCs; and (3) our axioms correspond to theirs. □

[3] We assume the standard relational projection semantics for $\pi_{\mathrm{attr}(Q)}$, i.e., we project on the attributes of Q.

Obtaining GO-FDs. GO-FDs encode domain knowledge of their scopes and are specified by a curator, typically with implicit knowledge of the graph's construction and the subgraphs relevant to the application. Skavantzos and Link [37] report that even for within-node dependencies, automated mining gives only heuristics; mining GO-FDs over more complex graph patterns is correspondingly harder, and automatic retrieval of GO-FDs remains an open problem.

5 Redundancies in Labeled Property Graphs

Since one of the main objectives of normalization is to reduce redundancy, we first identify different types of redundancies that might occur in LPGs under GO-FDs and show how these can be resolved in a graph-native way. Based on these insights, we introduce Graph-Native Normal Forms (GN-NFs).

5.1 Redundancy Patterns

Graph data is called *redundant if it represents the same piece of information multiple times.*

Example 5.1. Consider again the running example in Figure 1b and dependency $\varphi_{\mathrm{GroupNo}}$, which states that a student group's number determines the name of the group they are in. The fact that group 1 is named "Heroes" is represented twice, in edges e4 and e5. Figure 4a visually displays the redundancy of the group name in these edges. Preferably, the group name "Heroes" should only be represented once in combination with group number 1.

We can characterize these redundancies from a relational perspective. Consider the relation depicted in Figure 4b, which shows the relation represented by the scope of dependency $\varphi_{\mathrm{GroupNo}}$, including the source and target of the edges. Clearly, if we look at the attributes used in the descriptor of $\varphi_{\mathrm{GroupNo}}$, i.e., groupNo and name, we can identify the repeated fact.

To simplify our discussion of redundancies, we assume that the dependencies are *strictly* within or between objects, i.e., for a descriptor $X \Rightarrow Y$ at most one object identity variable may appear in each, X and Y. We call such dependencies *strict* GO-FDs. They capture natural dependencies, such as the ones used in our examples.

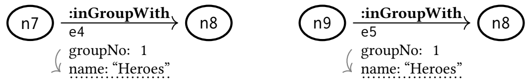
To aid our discussion of redundancy, we discuss the *shape* of a GO-FD: the shape is determined by inspecting the *descriptor* $X \Rightarrow Y$. Specifically, our discussion of redundancy relies on the types of variables that are present in X and Y (recall that variables are either identity- or property variables, cf. Section 2).

First, as shown in Figure 5, we identify three groups of GO-FDs for which no redundancies occur.[4] Each group contains a number of dependencies of different shapes, using the graphical notation. The source and target of the gray arrow indicate the shape. For simplicity, only one direction is shown for the edges. Still, the same classes also apply to the inverse direction.

Theorem 5.2. *No redundancies occur under dependencies that follow the shapes shown in Figure 5.*

Proof sketch. Groups (a)–(c) cover patterns for which the same fact cannot appear twice: (a) follows from the structurally implied rules (each edge has unique source/target; each property has at most one value); (b) are superkey constraints, which, by definition, do not allow for repetition; (c) constrains the out-degree to one. □

[4] While only one property key k is depicted in these dependencies, this is only a graphical simplification: generally, these can be sets of property keys.



(a) Redundancy within the edges

x	x.groupNo	x.name	$\hat{x}_{\text{src}}$	$\hat{x}_{\text{tgt}}$
e4	1	Heroes	n7	n8
e5	1	Heroes	n9	n8

(b) Redundancy from a relational perspective

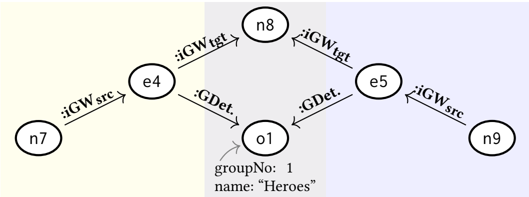
R_{edge}

x	x.groupNo	$\hat{x}_{\text{src}}$	$\hat{x}_{\text{tgt}}$
e4	1	n7	n8
e5	1	n9	n8

R_{groupNo}

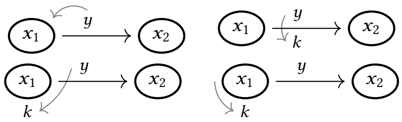
x.groupNo	x.name
1	Heroes

(c) Decomposed relations. The dashed line is a foreign key; the gray line is the descriptor of the original dependency.

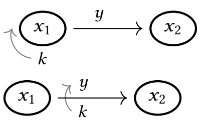


(d) Transformed subgraph of Figure 4a without redundancies

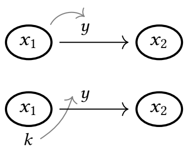
Figure 4: Resolving redundancies in Figure 1 related to φ_{GroupNo}: the original redundancy (a, b), its relational decomposition (c), and the transformed subgraph (d).



(a) Structurally implied GO-FDs



(b) Superkey GO-FDs



(c) Cardinality-limiting GO-FDs

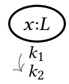
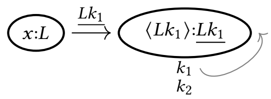
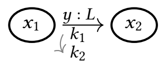
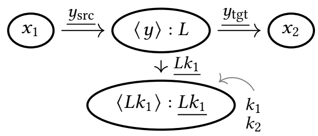
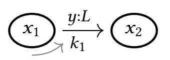
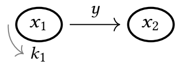
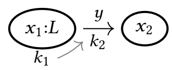
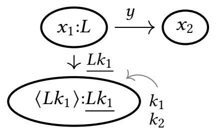
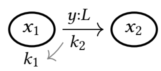
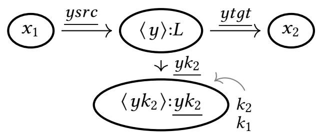
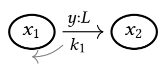
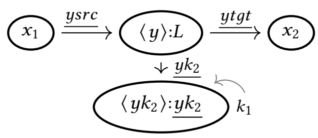
Figure 5: GO-FDs of different shapes for which no redundancies can occur. GO-FD descriptors are drawn in gray. For simplicity, only one direction of edges is shown.

All other dependency shapes allow for redundancies to occur. We call dependencies of such shapes *redundancy patterns*. They are listed in the second column of Table 4 (T).

5.2 Redundancy Removing Transformations

Since our objective is to eliminate redundant graph data, we propose a general technique that, for each possible redundancy pattern, eliminates the redundancy by specifying a specific graph transformation (cf. Section 2). This is similar to ER normalization [39], where also transformations are employed to avoid modeling that allows for redundancies. The transformations are

Table 4: Graphical representation of the graph transformations that resolve redundancy patterns. The descriptor of a GO-FDs is shown as a gray arrow. New nodes are constructed on values x matched in the redundancy pattern (shown using the $\langle x \rangle$ notation), similarly, new labels are constructed (shown using the $\underline{x}$ notation).

ID	Redundancy pattern (T)	Resolved redundancy pattern (T_{new})
Redundancies and transformations associated with within-graph object GO-FDs		
$\psi_{\text{within-n}}$		
$\psi_{\text{within-e}}$		
Redundancies and transformations associated with between-graph object GO-FDs		
$\psi_{\text{between-n-ep}}$		
$\psi_{\text{between-np-ep}}$		
$\psi_{\text{between-ep-np}}$		
$\psi_{\text{between-ep-n}}$		

shown in Table 4, which depicts the redundancy pattern (T), as well as the resolved pattern (T_{new}). Gray arrows denote the dependencies present in the patterns. For example, the transformation $\psi_{\text{between-n-ep}}$ applies to GO-FDs of the form

$$(x_1{:}\emptyset{:}\emptyset) \xrightarrow{y:L:\{k_1,\dots\}} () :: x_1 \Rightarrow y.k_1$$

Concretely, this covers, e.g., dependency φ_{Book} (cf. Section 2).

Generally, the discussed transformations follow a principled approach: elements mentioned in the descriptor $X \Rightarrow Y$ of a GO-FD belong together. More specifically, they are bundled together into a new node n_{new} that is identified by X. This node n_{new} additionally must be reachable from the graph object o of which X was initially a part of. If o is a node, an edge is created to reference n_{new} from o. However, when o is an edge, a different approach must be used.

For descriptors where the left-hand side consists of variables originating from an edge, we employ a *reification pattern*: the edge is replaced by a reifier node n_{reif}, which has an edge to the newly created node n_{new}. These strategies are summarized as graph transformations in Table 4. We motivate this approach with an example.

Table 5: Rationale for the redundancy removing transformations

ID	Relation schema of the redundancy pattern	Decomposed relation schema	Transformation rationale
Rationales for within-GO-FD transformations			
$\psi_{\text{within-n}}$	$\{x, x.k_1, x.k_2\}$	$R_1 = \{x, x.k_1\}$ $R_2 = \{x.k_1, x.k_2\}$ $x.k_1$ in R_1 is a foreign key (FK).	R_1 has one referencing attribute: the node x itself. We conclude that R_1 represents node x. R_2 has no referencing attributes, and therefore, it represents a new node n. Since R_1 has a FK to R_2, we materialize it as an edge between x and n. Since we no longer need the FK, k_1 needs not be stored in node x.
$\psi_{\text{within-e}}$	$\{x_1, x_2, y, y.k_1, y.k_2\}$	$R_1 = \{x_1, x_2, y, y.k_1\}$ $R_2 = \{y.k_1, y.k_2\}$ $y.k_1$ in R_1 is a FK.	R_1 has three referencing attributes: edge y, its source and target. We conclude that R_1 represents edge y. R_2 has no referencing attributes, and therefore, it represents a new node n. Since R_1 has a FK to R_2, we want to materialize it as an edge. Therefore, we reify y, with a reifier node n'. Then, we create the edge between n' and n. Since we no longer need the FK, k_1 needs not be stored in node n'.
Rationales for between-GO-FD transformations			
$\psi_{\text{between-n-ep}}$	$\{x_1, x_2, y, y.k_1\}$	$R_1 = \{x_1, x_2, y\}$ $R_2 = \{x_1, y.k_1\}$ x_1 in R_1 is a FK.	R_1 has three referencing attributes: edge y, and its source and target. We conclude that R_1 represents edge y. R_2 has one referencing attribute x_1, and therefore, we conclude it represents node x_1. Additionally, R_2 contains a new property key, which represents moving property key k_1 from y to x_1.
$\psi_{\text{between-np-ep}}$	$\{x_1, x_2, y, x.k_1, y.k_2\}$	$R_1 = \{x_1, x_2, y, x.k_1\}$ $R_2 = \{x.k_1, y.k_2\}$ $x.k_1$ is a FK.	R_1 has three referencing attributes: edge y, its source and target. We conclude that R_1 represents edge y. R_2 has no referencing attributes, and therefore, it represents a new node n. Since R_1 has a FK using a property from x_1, we materialize it as an edge between x_1 and n. Since we no longer need the FK, k_1 needs not be stored in node x_1.
$\psi_{\text{between-ep-np}}$	$\{x_1, x_2, y, x.k_1, y.k_2\}$	$R_1 = \{x_1, x_2, y, y.k_2\}$ $R_2 = \{y.k_2, x.k_1\}$ $X.x_1$ is a FK.	R_1 has three referencing attributes: edge y, its source and target. We conclude R_1 represents the edge y. R_2 has no referencing attributes, therefore, it represents a new node n. Since R_1 has a FK using a property key from y, we want to materialize it as an edge. Therefore, we reify y, with a reifier node n'. Then, we create the edge between n' and n. Since we no longer need the FK, k_1 needs not be stored in node n'.
$\psi_{\text{between-ep-n}}$	$\{x_1, x_2, y, y.k_1\}$	$R_1 = \{x_1, x_2, y, y.k_1\}$ $R_2 = \{x_1, y.k_1\}$ $y.k_1$ is a FK.	R_1 has three referencing attributes: edge y, and its source and target. We conclude that R_1 represents edge y. R_2 has one referencing attribute x_1, however, since it needs to be referenced from the edge y, an edge will be materialized between the reified node n' and a new node, specially for k_1.

Example 5.3. Continuing Example 5.1, we start by observing that relation R resulting from the scope of φ_{GroupNo}, as shown in Figure 4b, has three references to graph objects: edge x, its source and its target. This is characteristic for edges. Nodes, on the other hand, never refer to any other graph object.

We show that the redundant information discussed in Example 5.1 is eliminated by executing the transformation $\psi_{\text{within-e}}$ from Table 4. In the relational setting, we would decompose R into the two relations R_{Edge} and $R_{GroupNo}$ shown in Figure 4c. However, it is not clear from the outset how these two relations correspond to graph objects. Looking at $R_{GroupNo}$, it refers to no graph objects, which means it is a node in principle. On the other hand, R_{Edge} refers to *three* objects (the edge itself, source, and target), which corresponds to the edge x itself. Since R_{Edge} additionally also has a foreign key to $R_{GroupNo}$, we would like to capture this relationship by introducing an edge. However, we cannot create an edge where one of the endpoints is an edge itself. We argue that this must be resolved by introducing a *reifier* node ($\langle y \rangle$ in transformation $\psi_{\text{within-e}}$) that can reference nodes from $R_{GroupNo}$ using an edge. Furthermore, since we consider elements of $R_{GroupNo}$ to be nodes, they will need an identifier: in $\psi_{\text{within-e}}$ this identifier is constructed based on the labels and the property keys. Thus, we create an identifier based on the label inGroupWith and property key groupNo. Additionally, a suitable label l is constructed for this node based on the same label and property. The result after executing this transformation is shown in Figure 4d. Notably, while in relational decomposition $R_{GroupNo}$ the group number is a key, the transformed pattern represents this by adding an explicit key constraint, specifically for nodes with label l where the group number acts as a key for such nodes: $(x{:}l{:}\{\text{groupNo}, \text{name}\}) :: \text{groupNo} \Rightarrow x$.

We generalize the arguments from Example 5.3 in Table 5. Essentially, for each entry in Table 4, we look at the relation schema R induced by the redundancy pattern. In particular, we are interested in the number of attributes that reference graph objects. We call such attributes *referencing attributes*. Indeed, when the pattern is a node, there will be exactly one such attribute. Otherwise, if the pattern has an edge, there are three: the source, target, and edge identity. Given a descriptor $X \Rightarrow Y$ (indicated by the gray arrows in Table 4), we decompose R into two new relation schemas R_1 and R_2, with $\text{attr}(R_1) = \text{attr}(R) - Y$ and $\text{attr}(R_2) = X \cup Y$. Depending on the number of referencing attributes, we interpret the effect of the transformation, as described in the column "transformation rationale" of Table 5.

The redundancy removing transformations are both lossless and dependency preserving. Table 5 shows the equivalent relational decomposition for which we know these properties hold. Indeed, because the join of the decomposed relations schemas R_1 and R_2 results again in the relation schema of the redundancy pattern, we can say that the transformations are lossless. Furthermore, Table 4 specifies exactly in what way the original dependency is preserved (indicated by the grey arrow in the "Resolved redundancy pattern" column). Specifically, the left-hand side of the descriptor of the original dependency now acts as a key of the newly created nodes. To reconstruct this original dependency from the transformed graph, the (structurally implied) Object Identity reasoning rule needs to be applied.

Expressing the elimination of a redundancy pattern as a graph transformation provides insight into the elimination process itself. When new nodes are created in this process, they may need to be merged together. Recall, that the transformations introduce new identifiers based on existing values on the graph data. Indeed, we want these new identities to overlap in a controlled manner: this expresses the merging of nodes that express the same information.

Example 5.4. Figure 4d shows the subgraph of Figure 4a in which redundancy has been eliminated. The yellow highlight shows the application of the transformation on edge e4, and the blue highlight the transformation of e5. Since the identity of the newly created node o1 (representing the fact that group 1 is called "Heroes") is the same for edge e4 and e5, the reification nodes both point to o1.

5.3 Graph-Native Normal Forms

While each individual redundancy pattern can be resolved with its associated transformation, our goal is to mitigate all redundancy patterns described above. In this section, we define exactly under which conditions these redundancy patterns are mitigated. We call these the Graph-Native Normal Forms (GN-NFs).

In order to define the GN-NFs, we leverage the ideas of relational normal forms (cf. Table 2, [1, 18]), and ER diagram normalization (cf. Section 3, [39]). Recall the characteristics from the relational normal forms (cf. Table 2). We adopt the definitions of the first two graph-native normal forms, GN-1NF and GN-2NF, directly from their relational counterparts. We formulate them as:

Definition 5.5 (GN-1NF). A graph G is in *GN-1NF* if all property values are atomic, i.e., $\forall c_1, c_2 \in \mathbf{C} : c_1 \notin c_2$.

Indeed, GN-1NF is independent of the dependencies that are present. While – just as in many relational database systems – non-atomic values are supported in graph database systems, e.g., in form of a list datatype, these are typically not considered in classical approaches to normalization. Since our definition of LPGs does not consider non-atomic values, all LPGs satisfy GN-1NF by definition.

Definition 5.6 (GN-2NF). A graph G is in *GN-2NF* with respect to a GN-Schema Σ if every property mentioned in the schema is fully dependent on a key.

Recall the structurally implied axioms (cf. Section 4). Indeed, in any graph, for every property k of graph object o, there is a trivial key: the object o itself. Therefore, we conclude that every GN-Schema vacuously satisfies GN-2NF.

To define GN-3NF and GN-BCNF, we first introduce Q-GN-3NF and Q-GN-BCNF. We call these *scoped* normal forms, in line with our GO-FD terminology, since they are defined relative to a graph pattern Q (the scope).

Definition 5.7 (Q-GN-3NF). A graph G is in *Q-GN-3NF* with respect to a GN-Schema Σ if whenever $\Sigma \models Q :: X \Rightarrow Y$, then either: (1) X is a superkey for $\mathsf{attr}(Q)$, (2) Y is prime, i.e., it is a subset of a key, or (3) $Q :: X \Rightarrow Y$ is trivial.

Similarly, we define Q-GN-BCNF as follows:

Definition 5.8 (Q-GN-BCNF). A graph G is in *Q-GN-BCNF* with respect to a GN-Schema Σ if whenever $\Sigma \models Q :: X \Rightarrow Y$, then X is a superkey for $\mathsf{attr}(Q)$, or $Q :: X \Rightarrow Y$ is trivial.

These are direct adaptations of the well-known relational normal forms to the setting with LPGs and GO-FDs. Furthermore, they generalize the *L:P* normal forms of [37], which use node patterns for Q and strictly within-node dependencies in Σ.

Ultimately, we want to define a normal form for a GN-Schema without specifying a particular scope Q. In particular, we want a graph with a GN-Schema Σ to be in Q-GN-3NF or Q-GN-BCNF for every Q present in the dependencies of Σ.

Algorithm 1: The transformation-based approach to graph-native normalization based on the synthesis algorithm

Input: a scope Q; a schema Σ; and an LPG G

Phase 1: Determine the dependencies relevant to Q
$\Sigma_Q := \{Q :: D \mid Q' :: D \in \Sigma \text{ and } Q \sqsubseteq Q'\}$

Phase 2: Compute the minimal cover
$\Sigma_Q^{\mathsf{min}} := \mathtt{MinCov}(\Sigma_Q)$

Phase 3: Construct transformations and new dependencies
$\mathcal{T} := \{(Q, T_{new}) \mid (T, T_{new}) \in \mathcal{T}_{\mathsf{red}} \text{ s.t. } T \text{ matches } \varphi \in \Sigma_Q^{\mathsf{min}}\}$
$\Sigma_Q^{\mathsf{new}} := \mathtt{CreateDependencies}(\mathcal{T})$

Phase 4: Apply graph transformations
$G := \mathtt{Transform}(G, \mathcal{T})$

Result: Σ_Q^{new} is the schema and G is normalized w.r.t. Q

Definition 5.9 (GN-3NF and GN-BCNF). A graph G is in GN-BCNF (resp. 3NF) with respect to a GN-Schema Σ if, for every Q present in the dependencies of Σ, it is in Q-GN-BCNF (resp. 3NF).

Transforming a graph G that satisfies a schema Σ into a graph G' s.t. it satisfies Σ', which is in GN-BCNF (or 3NF), is the process of *graph-native normalization.* We say a graph (schema) is in Graph-Native Normal Form (GN-NF) if it is either in GN-3NF or GN-BCNF.

6 Graph-Native Normalization

Considering GO-FDs and GN-NFs, we are ready to define a graph-native approach for the normalization of LPGs. This normalization process considers not only within-node but also within-edge and, generally, between-graph-object dependencies. While Section 5 addressed each redundancy pattern individually, this section eliminates them collectively for a given schema, transforming any graph into GN-NF. We first define normalization with respect to a single scope Q (Algorithm 1), and then extend it to full normalization (Algorithm 2).

6.1 Scoped Normalization

We will start with the normalization process to transform a given graph G and GN-schema (i.e., a set of dependencies) such that, for a given scope Q, it satisfies Q-GN-3NF, and may satisfy Q-GN-BCNF.

The process is inspired by the *synthesis* algorithm of relational normalization [1], which always considers all dependencies. An alternative would be the decomposition algorithm from relational normalization [1], which guarantees BCNF but is not always dependency-preserving and could thus leave redundancies in normalized graphs. The process consists of four phases, shown in Algorithm 1.

Phase 1: Determining Applicable Dependencies. Since we normalize with respect to a particular scope Q, we need to determine which dependencies in Σ apply to that scope so that we can properly compute the minimal cover in the next phase. We know that a graph G satisfies a GN-Schema Σ, i.e., $G \models \Sigma$. Hence, by the definition of GO-FDs, the evaluation of the scope pattern Q in graph G, i.e., $[\![Q]\!]^G$, satisfies every functional dependency $X \Rightarrow Y$ of a GO-FD $Q :: X \Rightarrow Y$ that is present in Σ. However, these are not the only dependencies $[\![Q]\!]^G$ satisfies. We also need to consider every GO-FD whose scope Q' is more general than Q, i.e., $Q' \sqsupseteq Q$. Intuitively, this is because a more general scope Q'

also "covers" the matched subgraph patterns of the more specific scope Q. Formally, this is an application of the *Restriction* reasoning rule (cf. Section 4). Recall that Theorem 4.3 gives us a straightforward way to determine whether Q' is more general than Q.

Phase 2: Minimal Cover. We would like to have a minimal cover Σ_Q^{min} of the set of all dependencies that are relevant for the scope Q. As in relational normalization, this minimal cover is computed to avoid the consideration of redundant or non-minimal GO-FDs for the determination of transformations $\mathcal{T}$ in Phase 3. We say a GO-FD $\varphi : Q :: X \Rightarrow Z$ is (i) redundant if a dependency φ is implied by all other dependencies of a GN-schema Σ, i.e., $\Sigma - \{\varphi\} \models \varphi$, and is (ii) minimal if there is no $Y \subset X$ such that $\Sigma - \{\varphi\} \models Q :: Y \Rightarrow Z$. We use the set of applicable dependencies from Phase 1 (Σ_Q) to compute Σ_Q^{min}. After applying *Restriction* (cf. Section 4) to each dependency in Σ_Q, we can reason over them using Armstrong's Axioms and compute the minimal cover as for FDs [1].

Phase 3: Constructing Transformations. The core of the graph-native normalization approach are the redundancy-removing transformations from Section 5. We describe the transformation construction in detail in the extended version of this paper [33].

Phase 4: Applying Transformations. Finally, the graph is transformed according to the constructed transformations $\mathcal{T}$. Recall from Section 2 that a set of transformations is applied in two phases: new graph data is created first, then superfluous data is removed. Since each individual transformation is lossless and dependency-preserving, so is the combined application of $\mathcal{T}$.

Example 6.1. We show the application of Algorithm 1 (ScopedLPGNormalize) and highlight the importance of Phases 1 and 2. For this we consider a GN-Schema Σ, which relies on two scopes that use $P = \{\text{title}, \text{program}, \text{language}\}$:

Q_1: $(x{:}\{\text{Course}\}{:}P)$

Q_2: $(x{:}\{\text{Course}, \text{International}\}{:}P) \xrightarrow{y:\{\text{teaches}\}:\{\text{usingBook}\}} ()$

The considered GN-Schema Σ contains the dependencies:

φ_1: $Q_1 :: x.\text{title} \Rightarrow x.\text{program}$

φ_2: $Q_1 :: x.\text{title} \Rightarrow x.\text{language}$

φ_3: $Q_2 :: x.\text{program} \Rightarrow x.\text{language}$

φ_4: $Q_2 :: x.\text{language} \Rightarrow y.\text{usingBook}$

and φ_{Course} (cf. Example 4.2). For this application of Algorithm 1, we decide to perform a scoped normalization regarding scope Q_2 and schema Σ on graph a G.

The *first phase* loops over every GO-FD in Σ. Since Q_1 is more general than Q_2 (cf. Table 3), all GO-FDs that use Q_1 are added to Σ_Q. Intuitively, we can see that dependencies with scope Q_1 are applicable to Q_2: international courses are also courses, and if the necessary property keys are present, the (more general) course dependencies also apply to international courses. Trivially, Q_2 is more general than itself, so also every GO-FD that uses Q_2 is added to Σ_Q. However, dependency φ_{Course} is not applicable for scope Q_2 since it requires the presence of the property keys $\{\text{title}, \text{year}\}$, which is not a subset of P due to year.

Having collected all GO-FDs that hold for scope Q_2, *Phase 2* then computes the minimal cover of these dependencies. Even though φ_1, φ_2, φ_3, and φ_4 are present in Σ_Q, the minimal cover Σ_Q^{min} will not include φ_2, as it can be implied by the other three dependencies.

In *Phase 3* the graph transformations are determined: $\psi_{\text{within-n}}$ for both φ_1 and φ_3, and $\psi_{\text{between-np-ep}}$ for φ_4. Indeed, if we had kept φ_2 in Phase 3, the transformation resulting from that dependency would lead to the creation of a superfluous new node. Thus, Phase 2 prevents the creation of such redundant nodes.

Finally, in *Phase 4*, the transformations are applied to G.

Algorithm 2: FullLPGNormalize
Full normalization algorithm for transforming LPGs into GN-NF.

Input: a schema Σ; and an LPG G

Collect all scopes
$\mathcal{Q} := \{Q \mid Q :: X \Rightarrow Y \in \Sigma\}$

Topologically sort the scopes based on $\sqsubseteq$ (see Theorem 4.3)
$\mathcal{Q}_{\text{sorted}} := \texttt{Sort}_{\sqsubseteq}(\mathcal{Q})$

Normalize for each scope
for *scope* $Q \in \mathcal{Q}_{sorted}$ **do**
  | ScopedLPGNormalize(Q, G, Σ) *% see Algorithm 1*

Result: G is in GN-NF

6.2 Full Normalization

Algorithm 1 normalizes with respect to one scope, but GN-NF (Definition 5.9) requires normalization for every scope Q. *Full* graph-native normalization (Algorithm 2, FullLPGNormalize) achieves this by repeatedly applying ScopedLPGNormalize (Algorithm 1) to each scope. The order of application is crucial: scopes must be processed from least to most general to avoid side-effects. We therefore topologically sort the scopes in Σ using the "more general than" partial order ($\sqsupseteq$, cf. Theorem 4.3). The following example illustrates why more specific scopes must be considered first:

Example 6.2. Consider the situation from Example 6.1, where we normalized within the scope Q_2 only. We now normalize for all scopes and show the importance of the order of scopes for full normalization. Assume we consider the scope Q_1 first. This would transform every course according to the transformation rules. For example, by φ_1, the properties title and program would be moved from the course nodes to a new node due to transformation rule $\psi_{\text{within-n}}$. Specifically, this means that the *international* courses would also be affected by this transformation. However, as a side effect, this movement of properties makes the GO-FDs φ_3 and φ_4 no longer applicable. In particular, in this example, if we want to normalize with scope Q_2 after Q_1, transformations resulting from φ_3 and φ_4 would have no effect.

This is why the topological ordering is crucial. Executing normalization with the more specific scope Q_2 first avoids this situation. Then, in the beginning, only international courses will be modified, considering the dependencies relating to regular courses as well (due to the application of Restriction in Phase 1 of ScopedLPGNormalize). Afterward, normalization with the more general scope Q_1 only affects course nodes, as international courses have already been normalized and no longer match the original dependencies.

The result is a graph that is normalized with respect to Σ and every scope Q. Hence G is in GN-NF.

7 Evaluation

Technical Setup. We implemented our approach in Python utilizing an ANTLR grammar [30] to parse GO-FDs defined on graph patterns; it supports graphs stored in Neo4j and Memgraph. Our implementation and evaluation setup incl. Docker, artifacts,

Table 6: Definitions of evaluation metrics

Type	Name	Definition
Per graph G	$m_{\#\text{Node}}$	$\lvert N \rvert$
	$m_{\#\text{Edge}}$	$\lvert E \rvert$
	$m_{\text{AvgPropNode}}$	$\text{avg}\Big(\Big\{\big\lvert\{p \in \mathbf{P} : (n, p) \in \text{dom}(\text{prop})\}\big\rvert : n \in N\Big\}\Big)$
	$m_{\text{AvgPropEdge}}$	$\text{avg}\Big(\Big\{\big\lvert\{p \in \mathbf{P} : (e, p) \in \text{dom}(\text{prop})\}\big\rvert : e \in E\Big\}\Big)$
Per GN-Schema Σ	$m_{\#\text{Dep.sTotal}}$	$\lvert \varphi \in \Sigma \rvert$
	$m_{\#\text{Dep.sWithinNode}}$	$\lvert \varphi \in \Sigma$ where φ is a within-node dependency$\rvert$
	$m_{\#\text{Dep.sWithinEdge}}$	$\lvert \varphi \in \Sigma$ where φ is a within-edge dependency$\rvert$
	$m_{\#\text{Dep.sBetween}}$	$\lvert \varphi \in \Sigma$ where φ is a between-graph-object dependency$\rvert$
Per GO-FD φ	$m_{\text{MaxRed.Pot.}}$	$\max(\mathcal{G}_{count(D)}(\llbracket Q \rrbracket^G))$
	$m_{\text{AvgRed.Pot.}}$	$\text{avg}(\mathcal{G}_{count(D)}(\llbracket Q \rrbracket^G))$
	m_{Minim}	$\begin{cases} 1, & \text{if } \lvert\llbracket Q \rrbracket^G\rvert = 1 \\ \dfrac{\lvert\mathcal{G}_{count(D)}(\llbracket Q \rrbracket^G)\rvert - 1}{\lvert\llbracket Q \rrbracket^G\rvert - 1}, & \text{else} \end{cases}$
Query performance	$m_{\text{Q.Dur.}}$	Query execution time as returned by the graph system
	$m_{\text{Q.DBHit}}$	Database hits as returned by the graph system

Table 7: Graphs used in the evaluation

Graph	Source	$m_{\#\text{Node}}$	$m_{\#\text{Edge}}$	$m_{\text{AvgPropNode}}$	$m_{\text{AvgPropEdge}}$
London Public Transport	[34]	651	1952	≈ 5	3
Northwind	[27]	1035	3139	≈ 13.30	≈ 3.43
No Socks	[37]	3	2	≈ 3.33	0
Offshore	[28]	2 016 523	3 339 267	≈ 9.86	≈ 3.25
Train Services	[35]	194 179	1 826 601	≈ 6.98	≈ 3.79
University	Figure 1a	4	3	1.25	2

etc. are available on GitHub[5]. All experiments were run on a server with a 24-core 2.9 GHz AMD EPYC 9254 CPU and 768 GB DDR5 RAM. Neo4j was configured to use a page cache of 700 GB and a heap of 31 GB, as recommended by Neo4j.

Metrics. An overview of our metrics and their definitions are provided in Table 6. We distinguish between metrics measuring the structure and size of an LPG and metrics that measure query performance. Regarding graph structure and size, on a per-graph basis, we measure the number of nodes ($m_{\#\text{Node}}$) and edges ($m_{\#\text{Edge}}$), as well as the average number of properties per node ($m_{\text{AvgPropNode}}$) and edge ($m_{\text{AvgPropEdge}}$). For a set of GO-FDs Σ, i.e., a GN-Schema, we measure the number of all GO-FDs ($m_{\#\text{Dep.sTotal}}$) as well as within-node GO-FDs ($m_{\#\text{Dep.sWithinNode}}$), within-edge GO-FDs ($m_{\#\text{Dep.sWithinEdge}}$), and between-GO-FDs ($m_{\#\text{Dep.sBetween}}$).

In line with [37], we also calculate the 'potentials for redundancy' for each GO-FD. For a given GO-FD $Q :: X \Rightarrow Y$, and a graph G, the list of redundancy potentials M is calculated by a group-by-count query $M := \mathcal{G}_{count(X \cup Y)}(\llbracket Q \rrbracket^G)$, effectively counting how often the same combinations of values for property keys in X and Y occur. The higher the redundancy potentials (i.e., the counts in M) are, the more redundancy is present per value combination. If there are no redundant combinations of values, every value in M is 1. Based on M we calculate $m_{MaxRed.Pot.}$ as the maximum redundancy potential per GO-FD, and $m_{AvgRed.Pot.}$ as the arithmetic mean of the redundancy potentials M per GO-FD. Due to being based on counts, these two metrics do not have an upper limit.

Another structural metric we define per GO-FD is minimality $m_{\text{Minim.}}$ [14]. It complements the redundancy-potential-based metrics, as it is the ratio between the number of distinct and the total number of all combinations of values for property keys in X and Y. Thus, its result values lie between 0 (completely redundant, there is only a single combination of values for property keys in X and Y) and 1 (no redundancy at all, "minimal" in the sense that values for property keys in X and Y only appear in unique combinations).

For measuring query performance, we rely on the query execution time in milliseconds ($m_{\text{Q.Dur.}}$) and on the number of database hits ($m_{\text{Q.DBHit}}$), which is a graph system internal metric to express query effort. The absolute values for $m_{\text{Q.Dur.}}$ depend on the machine hosting the LPG system, $m_{\text{Q.DBHit}}$ is machine-independent. For both metrics lower values denote better performance.

Test scenarios. We evaluate our approach using six graphs (see Table 7 for details). To compare with related work, we use the same graphs as in [37] and add additional ones to evaluate additional types of dependencies. As shown in Table 8, they are used in 12 scenarios with different GN-Schemas that cover all transformations discussed in this paper (see Table 4). In lack of appropriate methods for mining constraints in graphs, we follow the approach described by [37] and manually define dependencies for each graph[6] and compute the minimal cover for each GN-Schema. The normalization of each test scenario was performed three times on fresh instances of Neo4j Enterprise, version 2025.12.1, and Memgraph 3.7.2. Query performance experiments were run ten times and the average of all runs is reported.

***Redundancy Reduction*.** Table 9 shows the results of our per-GO-FD metrics for each scenario before and after normalization. It shows that originally there are redundancies in all scenarios. After performing graph-native normalization, i.e., performing the redundancy removing transformations (cf. Section 5), all redundancies are eliminated. Thus, the per-GO-FD metrics $m_{\text{MaxRed.Pot.}}$, $m_{\text{AvgRed.Pot.}}$, and m_{Minim} all return 1, as for each dependency $\varphi : Q :: X \Rightarrow Y$ there is only a single combination of values for the property keys in X and Y.

***Effects on LPG Structure*.** Graph-native LPG normalization affects the structure and size of an LPG, with the precise impact depending on the transformations applied. Figure 6 shows the relative percentage change of our per-graph metrics on a logarithmic scale, broken down by dependency type: Figure 6a for all dependencies, Figure 6c for those leading to reification, and Figure 6b for those that do not. This reveals how different redundancy patterns (cf. Table 4) influence normalization.

Transformations that create new nodes without reification (e.g., $\psi_{\text{within-n}}$ and $\psi_{\text{between-np-ep}}$) introduce new nodes for the properties of the descriptor of a dependency and connect them to the existing subgraph via an edge. As Figure 6b shows, this increases the number of nodes and edges while decreasing the number of properties – an effect of storing each piece of information only once.

Transformations involving reification ($\psi_{\text{within-e}}$, $\psi_{\text{between-ep-n}}$, and $\psi_{\text{between-ep-np}}$) have the most significant effect on per-graph metrics. As Figure 6c shows, they also reduce the average number of properties but, due to reification, create substantially more new nodes and edges.

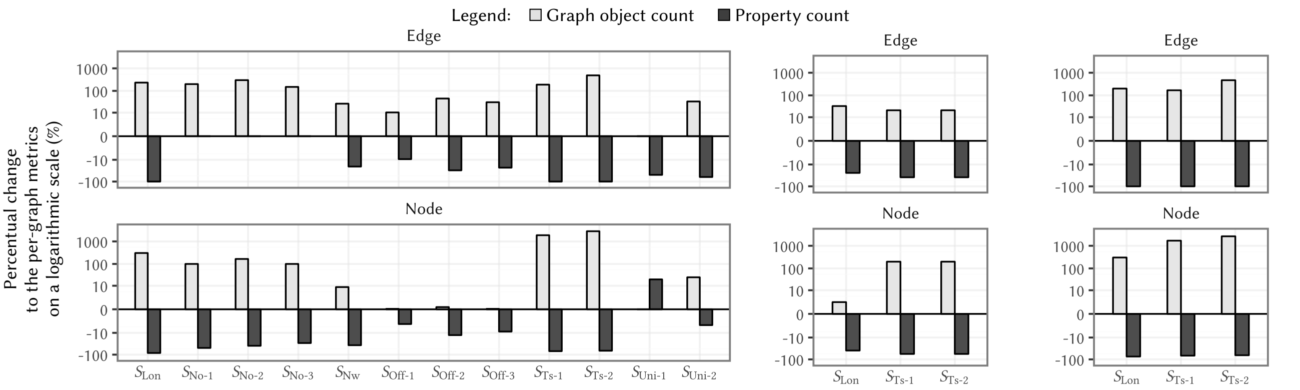
Overall, Figure 6a shows that graph-native normalization almost always increases the number of graph objects and decreases

[5] https://github.com/dmki-tuwien/lpg-normalization

[6] Details on the scenarios including the set of dependencies are available online: https://dmki-tuwien.github.io/lpg-normalization/evaluation_scenarios/

Table 8: Overview of the evaluation scenarios. Durations are in s and are rounded to two decimal places.

Scenario ID	Graph	*Provided GO-FDs* $m_{\#Dep.sW.\text{-}Node}$	$m_{\#Dep.sW.\text{-}Edge}$	$m_{\#Dep.sBetween}$	*Minimal cover of GO-FDs* $m_{\#Dep.sW.\text{-}Node}$	$m_{\#Dep.sW.\text{-}Edge}$	$m_{\#Dep.sBetween}$	Applicable transformations	Note	*Normalization duration* Memgraph	Neo4j
S_{Lon}	London Public Transport	3	2	0	2	1	0	$\psi_{within\text{-}n}$, $\psi_{within\text{-}e}$		0.11	1.33
S_{Nw}	Northwind	2	0	0	2	0	0	$\psi_{within\text{-}n}$	as in [37]	0.04	1.14
$S_{No\text{-}1}$	No Socks	4	0	0	4	0	0	$\psi_{within\text{-}n}$	as in [37]	0.03	1.17
$S_{No\text{-}2}$	No Socks	4	0	0	4	0	0	$\psi_{within\text{-}n}$	as in [37]	0.03	1.41
$S_{No\text{-}3}$	No Socks	5	0	0	5	0	0	$\psi_{within\text{-}n}$	as in [37]	0.02	1.45
$S_{Off\text{-}1}$	Offshore	3	0	0	2	0	0	$\psi_{within\text{-}n}$	as in [37]	9.18	24.77
$S_{Off\text{-}2}$	Offshore	5	0	0	5	0	0	$\psi_{within\text{-}n}$	as in [37]	33.92	116.28
$S_{Off\text{-}3}$	Offshore	10	0	0	5	0	0	$\psi_{within\text{-}n}$	as in [37]	25.15	63.26
$S_{Ts\text{-}1}$	Train Services	3	1	2	2	1	2	$\psi_{w.\text{-}n}$, $\psi_{w.\text{-}e}$, $\psi_{b.\text{-}np\text{-}ep}$, $\psi_{b.\text{-}ep\text{-}n}$		60.89	130.27
$S_{Ts\text{-}2}$	Train Services	3	1	3	2	1	3	$\psi_{w.\text{-}n}$, $\psi_{w.\text{-}e}$, $\psi_{b.\text{-}np\text{-}ep}$, $\psi_{b.\text{-}ep\text{-}n}$, $\psi_{b.\text{-}ep\text{-}np}$		104.16	408.72
$S_{Uni\text{-}1}$	University	0	0	1	0	0	1	$\psi_{between\text{-}n\text{-}ep}$	cf. Figure 1a	0.01	0.19
$S_{Uni\text{-}2}$	University	0	0	1	0	0	1	$\psi_{between\text{-}np\text{-}ep}$	adapted from Figure 1a	0.01	0.61



(a) All dependencies **(b) Without reification** **(c) With reification**

Figure 6: Percentual changes (log scale) to our per-graph metrics following graph-native LPG normalization grouped by graph object, considering all dependencies and those leading to transformations with and without reification.

Table 9: The results of the per-GO-FD metrics before and after graph-native LPG normalization

Scenario	avg($m_{MaxRed.Pot.}$) *before*	*after*	avg($m_{AvgRed.Pot.}$) *before*	*after*	avg(m_{Minim}) *before*	*after*
S_{Lon}	112.60	1	≈ 36.42	1	≈ 0.41	1
S_{Nw}	16.00	1	≈ 5.16	1	≈ 0.55	1
$S_{No\text{-}1}$	1.25	1	1.25	1	0.75	1
$S_{No\text{-}2}$	1.25	1	1.25	1	0.75	1
$S_{No\text{-}3}$	1.20	1	1.20	1	0.80	1
$S_{Off\text{-}1}$	≈ 264 881.33	1	≈ 613.07	1	≈ 0.00	1
$S_{Off\text{-}2}$	175 898.00	1	≈ 513.25	1	≈ 0.00	1
$S_{Off\text{-}3}$	239 572.20	1	≈ 21 645.63	1	≈ 0.00	1
$S_{Ts\text{-}1}$	≈ 6790.33	1	≈ 563.93	1	≈ 0.68	1
$S_{Ts\text{-}2}$	11 631.00	1	≈ 964.82	1	≈ 0.59	1
$S_{Uni\text{-}1}$	3.00	1	3.00	1	0.00	1
$S_{Uni\text{-}2}$	3.00	1	3.00	1	0.00	1

the average number of properties per object. The exception is $\psi_{between\text{-}n\text{-}ep}$, which "moves" a property from an edge to a node, as in $S_{Uni\text{-}1}$. There, $m_{\#Node}$ and $m_{\#Edge}$ remain constant while $m_{AvgPropEdge}$ decreases and $m_{AvgPropNode}$ increases. This reflects the movement of the property values from edges to unique nodes.

With respect to runtime, we observed three influencing factors: the number of graph objects before normalization, the number of dependencies in the minimal cover of the GN-Schema to consider, and the type of graph transformations applied during normalization. The latter correlates with the number of graph objects involved: transformations using reification are the slowest. For example, in S_{Lon} on Neo4j, the transformation of type $\psi_{within\text{-}n}$takes $\approx$ 0.58 s on average, while the transformation of type $\psi_{within\text{-}e}$ takes $\approx$ 0.97 s.[7]

Effects on Query Performance. Normalization changes the structure of an LPG and hence also affects query performance. We define 17 lookup and update queries across S_{Lon}, S_{Nw}, $S_{Off\text{-}1}$, and $S_{Ts\text{-}2}$ (cf. Table 8), classified along two axes. *Direct* queries retrieve property keys $k \in X \cup Y$ of a single GO-FD $\varphi : P :: X \Rightarrow Y$, i.e., information that is redundant in the original graph but unique after normalization; *indirect* queries consist of larger patterns. For direct queries, the normalized graph allows for two different query rewritings: an *optimal* rewriting that targets only the newly created nodes, and a *naive* rewriting which is non-optimal. Figure 7 reports $m_{Q.DBHit}$ before and after normalization for each query and its rewritings.

Direct queries with optimal rewriting show consistent reductions in database hits, independent of the type of transformation (including reification), and the type of query (lookup/update). Query runtimes ($m_{Q.Dur.}$) improve in a similar way as database hits ($m_{Q.DBHit}$). Naive rewritings and indirect queries show a

[7]Detailed results can be found at https://github.com/dmki-tuwien/lpg-normalization

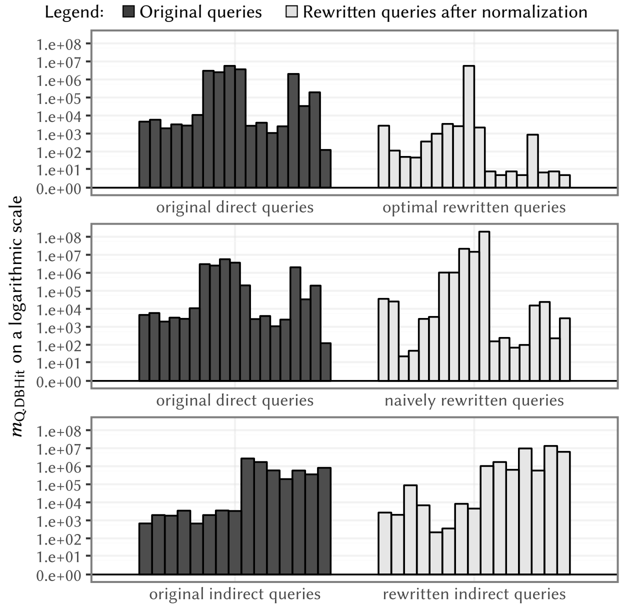


Figure 7: Comparison of query performance ($m_{\text{Q.DBHit}}$) between original and rewritten queries

Table 10: Comparison of per-graph metrics after normalizing using our approach and related work [37]

Scenario	Approach	$m_{\text{\#Node}}$	$m_{\text{\#Edge}}$	$m_{\text{AvgPropNode}}$	$m_{\text{AvgPropEdge}}$
$S_{\text{No-1}}$	Graph-Native	6	6	≈ 1.67	0
$S_{\text{No-2}}$	[37]	8	8	2	0
	Graph-Native	8	8	2	0
$S_{\text{No-3}}$	Graph-Native	6	5	≈ 2.33	0
S_{Nw}	[37]	1124	3969	≈ 8.30	≈ 2.71
	Graph-Native	1124	3969	≈ 8.30	≈ 2.71
$S_{\text{Off-1}}$	[37]	2 017 381	3 681 864	≈ 9.52	≈ 2.95
	Graph-Native	2 017 381	3 681 864	≈ 9.52	≈ 2.95
$S_{\text{Off-2}}$	[37]	2 021 602	4 854 231	≈ 8.60	≈ 2.24
	Graph-Native	2 021 602	4 854 231	≈ 8.60	≈ 2.24
$S_{\text{Off-3}}$	[37]	2 017 388	4 367 058	≈ 9.01	≈ 2.49
	Graph-Native	2 017 388	4 367 058	≈ 9.01	≈ 2.49

mixed picture: some benefit from reduced redundancy, others incur a cost from the larger patterns. The takeaway is that direct queries are a meaningful class since they represent exactly what users are likely to query: dependencies typically mark *important* information. Detailed queries and per-query results can be found in the extended version of this paper [33].

Comparison to Related Work. To show that our approach generalizes Skavantzos and Link [37], we compare the per-graph metrics on the scenarios supported by both works: $S_{\text{No-2}}$, S_{Nw}, $S_{\text{Off-1}}$, $S_{\text{Off-2}}$, and $S_{\text{Off-3}}$. As Table 10 shows, both approaches yield identical metrics, experimentally corroborating that $\psi_{\text{between-n}}$ is equivalent to the LPG node normalization of [37].

Scenario $S_{\text{No-1}}$ could in principle be covered by [37], but is omitted due to a subtle difference in the treatment of keys. In [37], gUCs are treated as dependencies that determine all property values of a node ($S_{\text{No-2}}$), whereas we treat keys as dependencies that determine a node's identity, which in turn determines its properties (cf. structurally implied dependencies, Section 4). As discussed in Section 5, such key dependencies do not cause redundancies and require no transformation, yielding a smaller – yet normalized – graph (visible when comparing $S_{\text{No-1}}$ and $S_{\text{No-2}}$). Scenario $S_{\text{No-3}}$ is mentioned in [37] but without results, so it is excluded from Table 10.

Discussion. As discussed above, we see that graph-native normalization extends the existing approach to LPG normalization and eliminates redundancies in LPGs. However, it also leads to an increase in the number of graph objects, i.e., nodes and edges, as Figure 6 shows. As an effect, the complexity of a graph's structure increases as well. While the total number of properties in a graph is reduced as an effect of redundancy reduction, it can happen that the average number of properties in nodes increases, due to reification and the movement of properties, as discussed for $S_{\text{Uni-1}}$. Thus, we conclude that optimizing for redundancy reduction and a minimal number of graph objects in a graph simultaneously is not possible.

However, with respect to query performance, normalization (and thus redundancy reduction) can lead to improvement. This is an interesting insight compared to the relational data model, where denormalized schemas are sometimes intentionally used to ease and speed up querying of relations.

Another difference between the relational model and LPGs concerns the definition of dependencies. LPGs provide a larger number of possible modeling choices compared to the relational model. Thus, LPG normalization depends more heavily on the initial structure of the data. This means that, compared to the relational model, more attention must be given to the design of the GO-FDs.

8 Conclusion

Building upon related work and in analogy with relational database systems, this paper proposes a graph-native approach to labeled property graph (LPG) normalization. The novel process employs graph transformation rules to normalize graphs into GN-NF and subsequently reduces redundancy. Our evaluation experimentally shows the applicability of our approach using a variety of graphs and functional dependencies; it highlights the tradeoff between having a simple structure and the reduction of redundancies, and it emphasizes the possible impact on query performance.

These observations open several avenues for future research. First, while we assume the availability of GO-FDs, in practice these dependencies may need to be determined automatically, e.g., through mining. Another aspect of future work is to consider a broader range of dependencies using complex graph patterns as their scope. Based on our insights on query performance we see the need for further research on rewriting queries for normalized graphs as well. Finally, we plan to investigate to what extent our work can be applied to knowledge graphs in RDF.

Acknowledgments

This project was partially funded by the European Union [Horizon Europe, TARGET, grant agreement no. 101136244] and the Vienna Science and Technology Fund (WWTF) [DORSET, grant id: 10.47379/ICT25032].

Artifacts

All artifacts associated with this work are available online on GitHub: https://github.com/dmki-tuwien/lpg-normalization

References

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

[2] Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Katja Hose, Maxime Jakubowski, José Emilio Labra Gayo, Wim Martens, Fabio Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savkovic, Mantas Simkus, and Dominik Tomaszuk. 2025. Common Foundations for SHACL, ShEx, and PG-Schema. In *WWW.* ACM, 8–21.

[3] 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.

[4] 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 Conference.* ACM, 2423–2436.

[5] William Ward Armstrong. 1974. Dependency Structures of Data Base Relationships. In *IFIP Congress.* North-Holland, 580–583.

[6] Carlo Batini and Monica Scannapieco. 2016. *Data and Information Quality - Dimensions, Principles and Techniques.* Springer.

[7] Angela Bonifati. 2025. Versatile Property Graph Transformations. *Proc. VLDB Endow.* 18, 12 (2025), 5516–5526.

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

[9] Antoon Bronselaer. 2021. Data Quality Management: An Overview of Methods and Challenges. In *FQAS (Lecture Notes in Computer Science, Vol. 12871).* Springer, 127–141.

[10] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[11] E. F. Codd. 1971. Further Normalization of the Data Base Relational Model. *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[12] E. F. Codd. 1974. Recent Investigations in Relational Data Base Systems. In *IFIP Congress.* North-Holland, 1017–1021.

[13] Claude Delobel and Richard G. Casey. 1973. Decomposition of a Data Base and the Theory of Boolean Switching Functions. *IBM J. Res. Dev.* 17, 5 (1973), 374–386.

[14] Lisa Ehrlinger and Wolfram Wöß. 2018. A Novel Data Quality Metric for Minimality. In *QUAT@WISE (Lecture Notes in Computer Science, Vol. 11235).* Springer, 1–15.

[15] 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.* ACM, 241–250.

[16] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL (Invited Talk). In *ICDT (LIPIcs, Vol. 255).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:22.

[17] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. 2014. Implication and Axiomatization of Functional Constraints on Patterns with an Application to the RDF Data Model. In *FoIKS (Lecture Notes in Computer Science, Vol. 8367).* Springer, 250–269.

[18] Andreas Heuer, Gunter Saake, and Kai-Uwe Sattler. 2018. *Datenbanken - Konzepte und Sprachen, 6. Auflage.* MITP.

[19] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2022. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (2022), 71:1–71:37.

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

[21] Ernests Lavrinovics, Russa Biswas, Johannes Bjerva, and Katja Hose. 2025. Knowledge Graphs, Large Language Models, and Hallucinations: An NLP Perspective. *J. Web Semant.* 85 (2025), 100844.

[22] Ulf Leser and Felix Naumann. 2007. *Informationsintegration - Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* dpunkt.verlag.

[23] Junhong Lin, Xiaojie Guo, Yada Zhu, Samuel Mitchell, Erik Altman, and Julian Shun. 2024. FraudGT: A Simple, Effective, and Efficient Graph Transformer for Financial Fraud Detection. In *ICAIF.* ACM, 292–300.

[24] Maude Manouvrier and Khalid Belhajjame. 2026. Graph functional dependencies: Analysis and translation to PG-schema. *Inf. Syst.* 136 (2026), 102633.

[25] Memgraph. 2026. *Graph data model.* https://memgraph.com/docs/data-modeling/graph-data-model

[26] Neo4j. 2026. Graph database concepts. https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/

[27] Neo4j. 2026. *Northwind Graph Example.* https://github.com/neo4j-graph-examples/northwind

[28] International Consortium of Investigative Journalists. 2026. Offshore Leaks Database. https://github.com/ICIJ/offshoreleaks-data-packages

[29] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. 1989. *The Structure of the Relational Database Model.* EATCS Monographs on Theoretical Computer Science, Vol. 17. Springer.

[30] Terence Parr. 2026. ANTLR. https://www.antlr.org

[31] Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. 2024. Transforming RDF Graphs to Property Graphs using Standardized Schemas. *Proc. ACM Manag. Data* 2, 6 (2024), 242:1–242:25.

[32] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. 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 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 (2021), 62–71.

[33] Johannes Schrott, Maxime Jakubowski, and Katja Hose. 2026. Graph-Native Normalization. arXiv:2603.02995

[34] Johannes Schrott, Maxime Jakubowski, and Katja Hose. 2026. *London Public Transport.* doi:10.5281/zenodo.18479731

[35] Johannes Schrott, Maxime Jakubowski, and Katja Hose. 2026. *Train Services.* doi:10.5281/zenodo.18478421

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

[37] Philipp Skavantzos and Sebastian Link. 2025. Third and Boyce-Codd normal form for property graphs. *VLDB J.* 34, 2 (2025), 23.

[38] Philipp Skavantzos, Kaiqi Zhao, and Sebastian Link. 2021. Uniqueness Constraints on Property Graphs. In *CAiSE (Lecture Notes in Computer Science, Vol. 12751).* Springer, 280–295.

[39] Bernhard Thalheim. 2000. *Entity-relationship modeling - foundations of database technology.* Springer.

[40] Bernhard Thalheim. 2020. Schema Optimisation Instead of (Local) Normalisation. In *FoIKS (Lecture Notes in Computer Science, Vol. 12012).* Springer, 281–300.