Structural Normalization of Property Graphs

Maximilian K. Egger
maximilian.egger@cs.au.dk
Aarhus University

Mehdi Allali
mehdi.allali1@etu.univ-lorraine.fr
University of Lorraine

Matteo Lissandrini
matteo.lissandrini@univr.it
University of Verona

Davide Mottin
davide@cs.au.dk
Aarhus University

Panagiotis Karras
piekarras@gmail.com
U. of Copenhagen, Aarhus U.

ABSTRACT

A *property graph* stores information in the form of a network of nodes and edges, annotated with labels and properties (i.e. attributes). Even though various forms of functional dependencies, uniqueness constraints, and keys have been proposed for graph data, no study has been hitherto undertaken that builds upon and utilizes these concepts to assemble a comprehensive proposal for *graph normalization*. An elementary proposal merely maps the nodes in the graph schema to a relational schema and deploys relational normalization techniques. Unfortunately, this proposal disregards the graph structure and thereby forfeits the opportunity for *structural* normalization. In this paper, we define five structural graph normal forms that extend relational normal forms building on concepts of keys and graph functional dependencies. Based on acyclic graph patterns, these normal forms apply to any data graph, address previously overlooked sources of inconsistency, and support structure-aware integrity preservation beyond node attributes. We showcase the applicability and effectiveness of these normal forms through experimentation.

PVLDB Reference Format:
Maximilian K. Egger, Mehdi Allali, Matteo Lissandrini, Davide Mottin, and Panagiotis Karras. Structural Normalization of Property Graphs. PVLDB, 19(6): 1400 - 1412, 2026.
doi:10.14778/3797919.3797943

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AU-DIS/GraphNormalization.

1 INTRODUCTION

Normalization brings databases to a form that avoids inconsistencies and allows preserving dependencies among attribute values via *integrity constraints*. Those dependencies are expressed as *functional dependencies* (FDs) among sets of attributes and help detect keys, i.e., sets of attributes that identify entities. In relational databases, normalization follows a consolidated hierarchy of progressively more restrictive normal forms [11]. A database in *first normal form* (1NF) does not have any relation as attribute value; *second normal form* (2NF) enforces dependence of non-key attribute values on the *whole* key, i.e., eliminates *partial* dependencies, while *third normal form* (3NF) prescribes dependence of non-key attribute values on *nothing but* the key, i.e., eliminates *transitive* dependencies.

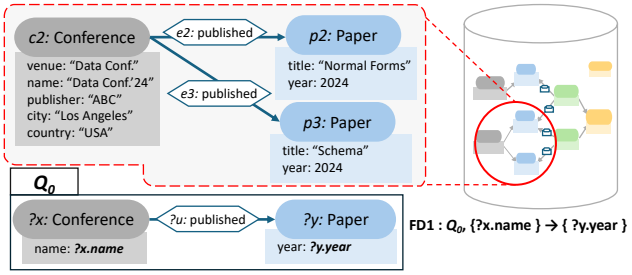

Figure 1: A section of a graph with a graph pattern Q and a corresponding functional dependency FD1.

Due to its effectiveness in reducing redundancy, avoiding anomalies, and preserving data integrity, normalization has been extended to semi-structured data repositories such as object stores [24, 27] and XML databases [6–8]. XML normalization, for instance, requires functional dependencies forming paths in a tree [6–8]. However, normalization has yet to be satisfactorily extended to schemata for graph databases, i.e., knowledge graphs [17, 21] and property graphs [3, 4], which store properties and interrelationships among entities. The introduction of the *Graph Query Language Standard* (GQL) [2], which forms the foundation for unified graph database modeling, has made the need for a normalization framework even more urgent. Recent research established the preliminaries for defining graph normalization. An initial proposal focuses on the graph structure and defines functional dependencies [14–16] and keys [13] as graph patterns. Another attempt defines keys and uniqueness constraints as combinations of properties and node labels [4]. Such works provide expressive frameworks to describe integrity constraints and dependencies, yet refrain from addressing the graph normalization problem. A recent proposal [25] approached the graph normalization problem by translating nodes in a property graph into entries in a relational table. Unfortunately, this simple approach disregards graph functional dependencies and rather relies on node uniqueness constraints [26]. By exclusively considering nodes, it overlooks redundancies that propagate over edges. For instance, it neglects the redundancy of storing $y.year$ in paper nodes instead of conference nodes in the graph of Figure 1. This recent work, on the one hand, motivates the need of graph normal forms, on the other hand proves the limits of simply transposing relational normalization over to graph, i.e., relational normalization can only extend to node-as-relation model and completely misses normalization opportunities involving edges.

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797943

In this paper, we formulate a holistic, theory-guided solution to the graph normalization problem, introducing five *graph normal forms* (GNF) that consider the entire graph structure encompassing edges and edge properties. These GNFs treat nodes and edges as first-class citizens by explicitly representing them as nodes and edges that would otherwise be implicitly modeled as properties. This novel formulation addresses the deficiencies of previous work on graph normalization and has no counterpart in the relational model, where it would entail moving attributes across relations. While we focus on the property graph model [3], the framework also applies to multi-layer graphs [5], Amazon OneGraph [22], and MillenniumDB [28] as long as their data can be expressed as a property graph. Hence, in this work:

(1) We re-frame the existing theory on graph functional dependencies for the scope of graph normalization, making it able to capture edge attributes as well as redundancies laying across nodes within a graph pattern, and we thus identify the limitations of existing solutions.
(2) We propose an array of *five* increasingly more restrictive graph normal forms (GNFs), each addressing a cause of redundancy: 1GNF disallows implicit links between nodes that may be regular edges and nested attributes that obscure data complexity; 2GNF disallows data replication caused by attributes depending on the key of another node; 3GNF disallows attributes depending only partially on a node's key; 3^+GNF only allows attributes directly dependent on a superkey; EGNF eliminates all value duplication possibilities by making a key out of any property.
(3) We evaluate how the proposed GNFs reduce data duplication compared to prior work, while preserving competitive query performance on the normalized graph.

2 FOUNDATIONS

Here we discuss the foundations for graph normalization, including *graph functional dependencies* and *keys for graphs* (Section 2.1) and *normal forms* for relational and XML data (Section 2.2).

2.1 Graph keys and dependencies

Keys uniquely identify a specific entity in a database. In graphs, keys uniquely identify nodes [4, 13]. In a KG [13], the graph is modeled via nodes and labeled edges without properties, i.e., literal values are also nodes; thus a key is typically a subgraph that appears only once in the graph. Equivalently, KG keys are queries that return exactly one node for a given variable assignment. Keys in *property graphs* (PG-Keys) [4] are still patterns but further include node and edges properties. Graph-tailored unique constraints (gUCs) [26] are special cases of PG-keys that only admit node properties.

Functional dependencies (FDs) define one-way relationships among attributes in a data set. As such, they constitute building blocks for keys and normal forms [11]. Graph functional dependencies (GFDs) tailor the intuition behind FDs for relational data to graphs [14–16, 25] and to the specific graph model, such as property graphs, multi-labeled graphs, and knowledge graphs. GFDs for KGs [16] are pairs comprising a graph pattern $Q[\bar{x}]$ with variables $\bar{x}$ and an implication $X \rightarrow Y$ between two sets of literals X,Y in $\bar{x}$, while graph association rules (GARs) [15] extend GFDs with edge literals and possibly a classifier for edge prediction. The graph pattern in a GFD sets the scope of the dependency, as the schema does in a relation. However, as GFDs do no limit the structure of that pattern, determining whether a graph complies with a GFD is computationally intractable. On the other hand, *graph-tailored functional dependencies (gFDs)* [25] are ill-suited for property graph normalization, as a gFD only captures dependencies between the attributes of a single node. Contrariwise, our formulation incorporates node and edge properties alongside the graph structure.

2.2 Data normalization

In **relational databases** normal forms (NF) [11] progressively reduce redundancies in data representation. The *first normal form (1NF)* prohibits attributes whose domain is a relation; every attribute should be atomic, i.e., indivisible. The respective normalization creates a new relation with the composite attribute decomposed into atomic ones. The *second normal form (2NF)* precludes non-prime attributes (i.e., those that are not part of a key) from functionally depending on a proper subset of a candidate key. The respective normalization isolates each set of such *partially* dependent non-prime attributes into their own relation with their determinant attributes as key. The *third normal form (3NF)* bars non-prime attributes from *transitively* functionally depending on a key. The normalization creates a new relation for each set of such attributes and their determinants. By the *Boyce-Codd normal form (BCNF)*, all attributes should functionally depend only on the key. There are of course more normal forms, but we focus here on these four as they are those that address the most fundamental sources of data anomalies.

XML normalization [6–8] extends BCNF to tree-structured XML documents, identifying two types of violation. The first type pertains to document structures with attribute values repeated across nodes; the normalization creates a new tree node that gathers the attributes involved in the violation. The second type avoids attribute redundancies among a node and its descendants; the normalization moves the redundant attributes from children to fathers.

Graph normalization. A preliminary approach to property graph normalization [25] builds upon uniqueness constraints [26] and *graph-tailored functional dependencies* [25] that target node properties. The normal forms defined over these functional dependencies are a direct transformation of graph nodes to relations and thus a mapping to 3NF and BCNF in the relational model when considering only subsets of nodes and their properties and ignoring edges altogether. Contrariwise, our normal forms embody dependencies among edges, nodes, and their properties and capture dependencies holding across graph-structures. We discuss how our proposals supersede previous work in Section 4.

3 FORMALIZATION

We now lay the foundation for the normalization of *property graphs*, which assign values to selected properties for each node and edge. We introduce graph functional dependencies and associated keys [14–16], based on flexible and complex graph structural patterns, which we use to define graph normal forms in Section 5. Table 1 gathers the notations we employ.

Definition 1 (Record [3]). *Given countable sets of attribute names $\mathcal{K}$, attribute values $\mathcal{V}$, a* ***record*** *is a finite-domain partial*

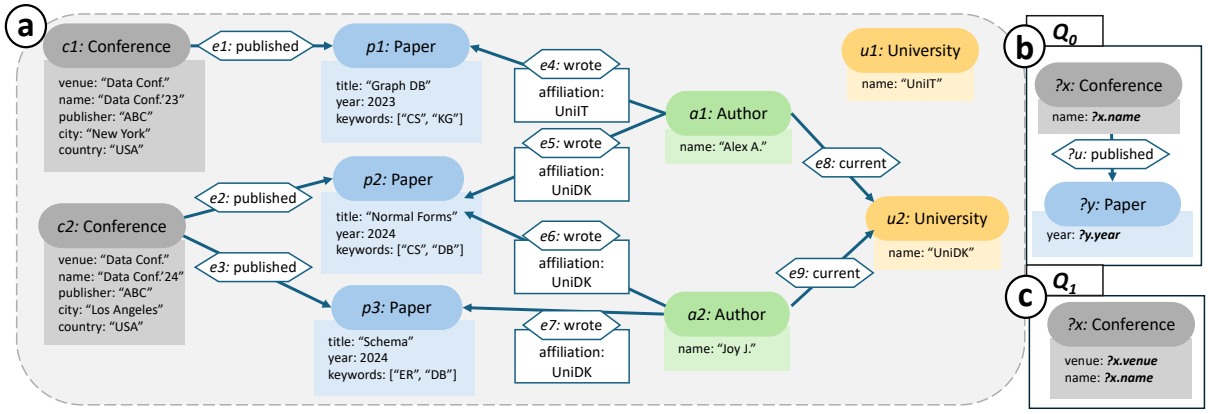

Figure 2: Example of non-normalized graph and two graph patterns.

Table 1: Notations.

Notation	Meaning
$G{=}(N, E, \rho, \ell, \Phi)$	property graph
$N/E/\mathcal{L}/\mathcal{R}/\mathcal{K}/\mathcal{V}$	set of nodes/edges/labels/records/keys/values
ρ	mapping of edges to pair of nodes
ℓ	mapping nodes/edges to $\mathcal{L}$
Φ	mapping of nodes/edges to records
$g{\in}N{\cup}E$	graph element: node or edge of G
$\Phi(g)$	set of record keys (i.e., attributes) of graph element g
$\mathcal{X}$	set of variables in a pattern
A, B, S	set of attribute literals
$Q[\mathcal{X}] = (N_Q, E_Q, \rho_Q, \ell_Q, \Phi_Q, \mu)$	graph pattern
μ	mapping of $\mathcal{X}$ to nodes / edges / values
$\mathcal{F} = (Q[\mathcal{X}], A{\mapsto}B)$	graph functional dependency
$(Q[\mathcal{X}], S)$	graph key
$k(g)$	key for a specific graph element

function $o{:}\mathcal{K}{\to}\mathcal{V}$ mapping names in $\mathcal{K}$ to values in $\mathcal{V}$. We use $\mathcal{R}$, to denote the set of records, and given a name-value pair (k, v) for record r, k is an attribute name of r with attribute value v.

Definition 2 (Property Graph [3]). *A **property graph** is a tuple $G{=}(N, E, \rho, \ell, \Phi)$ where:*

- *N is a finite set of nodes;*
- *E is a finite set of edges such that $N \cap E = \emptyset$;*
- *$\rho : E{\to}(N{\times}N)$ is a total function mapping edges to pairs of nodes;*
- *$\ell : (N \cup E){\to}2^{\mathcal{L}}$ is a total labeling function mapping nodes and edges to finite sets of labels (including the empty set);*
- *$\Phi : (N \cup E){\to}\mathcal{R}$ is a function mapping nodes and edges to records, and thus assigning attributes (also called properties) to nodes and edges.*

Definition 3 (Graph Element). *Given a graph $G{=}(N, E, \rho, \ell, \Phi)$, a **graph element** g is $g{\in}N{\cup}E$.*

By slightly abusing notation, we use $g{\in}G$ to denote $g{\in}N{\cup}E$ and $\Phi(g)$ to denote the set of record keys (i.e., attribute names) of graph element g. Normalization aims to eliminate redundancies emerging from *graph elements* storing properties as key-value pairs.

Graph pattern. A graph pattern is a graph in which nodes, edges, and attribute values function as *variables* defining relationships among graph elements and attributes.

Definition 4 (Graph Pattern). *Let $\mathcal{X}$ be a finite set of variables. A **graph pattern** (or query) is a tuple $Q[\mathcal{X}] = (N_Q, E_Q, \rho_Q, \ell_Q, \Phi_Q, \mu)$ where: $(N_Q, E_Q, \rho_Q, \ell_Q, \Phi_Q)$ is a property graph and $\mu{:}\mathcal{X}{\to}N_Q{\cup}E_Q$ is a partial bijective function mapping the variables $\mathcal{X}$ to nodes and edges in the pattern.*

Graph patterns, as in GQL [2], specify a subgraph to match; unspecified attribute values are unconstrained and thus still match.

Definition 5 (Pattern Literal). *A pattern literal $x.a$ for a variable $x{\in}\mathcal{X}$ of a pattern $Q[\mathcal{X}] = (N_Q, E_Q, \rho_Q, \ell_Q, \Phi_Q, \mu)$ is the attribute corresponding to the key a in the record $\Phi_Q(\mu(x))$*

The *matches of Q* are subgraph isomorphisms[1]. A graph H is *subgraph-isomorphic* to a graph G if there exists an injective mapping $f : V(H){\to}V(G)$ such that for every node $u \in V(H)$, $\ell_Q(u){=}\ell_Q(f(u))$ and $\Phi_Q(u){=}\Phi_Q(f(u))$, and for every edge $(u, v){\in}E(H)$, $(f(u), f(v)){\in}E(G)$ with $\ell_Q((u, v)) = \ell_Q((f(u), f(v)))$ and $\Phi_Q((u, v)) = \Phi_Q((f(u), f(v)))$. Thus, all specified labels and attributes must be preserved, while unspecified attributes and labels may vary. We call the nodes, edges, and attribute values corresponding to variable mappings *assignments*. Finally, $Q[\mathcal{X}]{\sqsubseteq}Q_e[\mathcal{X}_e]$ denotes an extension of $Q[\mathcal{X}]$, where Q is a subgraph of Q_e and $\mathcal{X}{\subseteq}\mathcal{X}_e$.

Example 1. *In Figure 2b, graph pattern Q_0 matches every paper y and the conference x that published it; literal $x.name$ is the conference name, and $y.year$ the year of the paper.*

Graph functional dependencies (GFDs). We use the above definitions to define graph functional dependencies as relationships among the literals of variables in a graph pattern.

Definition 6 (Graph Functional Dependency (GFD)). *A graph functional dependency $\mathcal{F}$ is a pair $(Q[\mathcal{X}], A{\mapsto}B)$, where Q is a graph pattern with variables $\mathcal{X}$ and A, B are sets of pattern literals for graph elements mapped to variables in $\mathcal{X}$.*

A graph functional dependency $(Q[\mathcal{X}], A{\mapsto}B)$ indicates that if the variable assignments in two subgraphs matching Q assume the same assigned values for literals in A, then they also assume the same assigned values for literals in B. We say that an attribute $a \in \mathcal{K}$ is **functionally dependent** on A by $\mathcal{F}$ if $\mathcal{F} = (Q[\mathcal{X}], A{\mapsto}a)$.

Example 2. *Given graph pattern Q_0 in Figure 2b, the GFD $(Q[\mathcal{X}], \{x.name\}{\mapsto}\{y.year\})$ implies that all papers in the same edition of the same venue (i.e., sharing values on $x.name$) were published in the same year (i.e., also share values for $y.year$).*

[1]Graph query languages use different homomorphism variants; we use isomorphism to preserve both structure disjointedness and edge/node values correspondence.

We note that, whereas in a relational database, an FD applies to attributes within a relation, a GFD may involve attributes (pattern literals) associated with different elements in the graph pattern. Consider for example the pattern Q_0 in Figure 2b, where the conference name fully determines the paper year. Because of this, the number of GFDs one can consider can be exponentially large as it grows with the entire graph. Yet, not all of these patterns are informative, and evaluating some of them can become quickly unfeasible. Thus, while GFDs, as defined above, allow arbitrary patterns, we formally restrict the set of GFDs admissible in our analysis to ensure tractability in the resolution of GFDs. In practice, to make normalization feasible, we focus only on FDs that describe relationships either between attributes within the same graph element or between pairs of graph elements linked by a simple acyclic path. While we believe that the study of more complex GFD structures and their computability bounds are important future work, we will see later that the current definition can already cover the most important causes of redundancies.

Definition 7 (Applicable GFD). *A GFD* $(Q[\mathcal{X}], A \mapsto B)$ *is* applicable *if it fulfills all of the following:*

- $Q[\mathcal{X}]$ *is an acyclic simple path when ignoring edge directions.*
- $\exists x, y \in \mathcal{X}$ *s.t.* $A \subseteq \Phi(x)$, $B \subseteq \Phi(y)$.

In what follows, whenever we write GFD, we specifically mean an applicable GFD.

Graph keys[2]**.** By analogy to relational normalization, a *superkey* is a subset of attributes connected by a graph pattern that uniquely identifies the attribute values for a set of graph elements; a *candidate key* is a minimal superkey, i.e., such that any subset of its attributes does not form a superkey. It follows that two graph elements sharing the same values in all superkey attributes should share the same values in the other non-key attributes of the graph pattern.

Definition 8 (Superkey and Candidate Key). *A pair* $(Q[\mathcal{X}], S)$ *is a* ***superkey*** *for a set of assignments (graph elements)* $\bar{G} \subseteq N_Q \cup E_Q$ *of a variable* $x \in \mathcal{X}$ if *for each* g *in* $\bar{G}$ *with attribute set* $\Phi(g)$*, there holds a functional dependency* $(Q[\mathcal{X}], S \mapsto \Phi(g))$*. Intuitively, this means that if two matches of* $Q[\mathcal{X}]$ *assign the* same values *to all variables in* S*, then they must induce the* same attribute values *on every attribute in* $\Phi(g)$*. A* ***candidate key*** *for* $\bar{G}$ *is a* minimal *superkey, i.e., one such that there exists no superkey* $(Q[\mathcal{X}], S')$ *for* $\bar{G}$ *with* $S' \subset S$.

We denote the set of candidate keys for a set of graph elements $\bar{G}$ as $k(\bar{G})$ and by extension, we also denote the set of candidate keys for a single graph element $g \in \bar{G}$ as $k(g)$.

Example 3. *Given the graph pattern in Figure 2c, the values of conference venue and conference edition name uniquely identify a conference node. Hence,* $(Q_1[\mathcal{X}], \{x.name, x.venue\})$ *are a superkey for conference nodes, i.e., for mappings of variable* x*. Yet, since the conference edition name is already enough,* $(Q_1[\mathcal{X}], \{x.name\})$ *is the corresponding candidate key. .*

Similar to relational databases it can occur that a set of attributes for some graph elements are also corresponding to attributes that are key to another element, we define these as foreign keys.

[2]Graph keys should not be confused with record keys.

Definition 9 (Foreign key). *Given two graph patterns* $Q[\mathcal{X}]$ *and* $Q'[\mathcal{X}']$ *with a non-empty overlap* $Q[\mathcal{X}] \sqcap Q'[\mathcal{X}'] \neq \emptyset$ *and a set of attributes* F*,* $(Q[\mathcal{X}], F)$ *is a foreign key towards superkey* $(Q'[\mathcal{X}'], S)$ *if given the joinable union* $Q[\mathcal{X}] \sqcup Q'[\mathcal{X}']$ *along the common graph elements it holds the functional dependency* $(Q[\mathcal{X}] \sqcup Q'[\mathcal{X}'], F \mapsto S)$ *and that value assignments for variables in* F *correspond to identical assignment in* S.

Our definition of GFDs specializes previously proposed Graph Association Rules (GARs) and Graph Functional Dependencies [14, 16], where a graph pattern may assume an arbitrary structure. We note that with arbitrary structures, the enforcement of functional dependencies during normalization can lead to ambiguities in resolving attribute values. We avoid such repercussions with our definition of GFDs to only consider acyclic path patterns $Q[\mathcal{X}]$ ignoring edge directions; since this renders the underlying matching problem computationally tractable and still enables expressive dependencies. We exclude DAG-shaped patterns because they introduce ambiguities in resolving GFDs. When multiple paths converge, it becomes unclear which direction the dependency resolution should follow, and symmetric structures lead to automorphisms that prevent a unique mapping of nodes and edges. Addressing these cases is non-trivial, and we leave such extensions to future work. This restriction also excludes some GFDs built on keys for weak graph elements, namely graph elements which are uniquely identified through neighbors, and dependencies that rely on multiple other elements, i.e., the attributes in the implication are sourced from multiple elements. Our approach cannot uniquely resolve such cases, due to the presence of multiple candidate locations for attribute relocation.

Some GFDs linking attributes of different graph elements are trivially true if a graph element unequivocally determines a connected graph element. If there exists a GFD $(Q[\mathcal{X}], A \mapsto B)$ with $A = k(x)$ and $B = k(y)$ for any $x \neq y \in \mathcal{X}$, then any GFD $(Q[\mathcal{X}], A \mapsto C)$ with $C \subseteq \Phi(y)$ is trivially true. In the following, we ignore such GFDs as they provide no useful information in normalizing graphs.

Finally, graph normalization requires integrity constraints to enforce dependencies that span across edges and even paths of two or more edges. Relational normalization, instead, consider a single relation at-a-time. For example, graph normalization would need to consider the dependency between a paper's *year* and its *conference* in Figure 4, which would translate to FDs spanning, for example, an equivalent set of relations. Graph normalization gracefully enforces the dependency through the graph structure.

4 RELATIONSHIP WITH gFDS AND gUCS

Before we embark on our journey to define graph normal forms, we need to establish how the graph functional dependencies in Definition 6 and the keys in Definition 8 relate to the recently proposed gFDs and gUCs [25]. We provide a formal argument in terms of expressiveness (Section 4.1) and axiomatization (Section 4.2).

4.1 Expressiveness

Our graph functional dependencies (Definition 6) restrict previously defined functional dependencies for graphs [16] to acyclic graphs, while being more general than the recently proposed gFDs [25]. Any gFD corresponds to GFD, while the opposite does not hold, since

gFDs cannot specify nodes connected with edges. The following theorem formalizes this relationship.

Theorem 1. *Both gFDs and gUCs servings as keys [25] are graph functional dependencies (GFDs) on single-node graph patterns, by Definition 6. In reverse, any GFD on a single-node graph pattern is equivalent either to a gFD L:P:X or to a gUC.*

Proof. A gFD $\sigma = L{:}P{:}X \rightarrow Y$ with $L \subseteq \mathcal{L}$, $P \subseteq \mathcal{K}$, and $X, Y \subseteq P$ corresponds to a GFD $\mathcal{F}_\sigma = (Q_\sigma[\mathcal{X}] = (N_\sigma, E_\sigma, \rho_\sigma, \ell, \Phi_\sigma, \mu), A_\sigma \mapsto B_\sigma)$, where $A_\sigma{=}X$, $B_\sigma{=}Y$, and the graph pattern is a single node without edges, i.e., $E_\sigma = \rho_\sigma = \emptyset$. The pattern restricts ℓ_σ to a specific set of labels L and lets Φ_σ only map to records that use keys in P. Similarly, a gUC $\phi = L{:}P{:}X$ with labels $L \subseteq \mathcal{L}$, properties $P \subseteq \mathcal{K}$, and $X \subseteq P$, which denotes that a node with labels L and properties P is uniquely determined by its values in X, corresponds to a GFD on a single-node graph pattern with $A_\sigma{=}B_\sigma{=}X$. □

4.2 Axiomatization

Here, we provide an axiomatic construction as a set of primitive truth statements to derive new graph functional dependencies to a given set of graph functional dependencies. The axiomatization does not rely on the acyclicity constraint and thus remains applicable in a more general setting. Figure 3 presents the six axioms referring to literals, marked in darker (blue) color, and thus considering only graph patterns for single nodes [25], and the three axioms we introduce marked in a lighter (green) color that cover instead proper graph patterns. In each axiom, the premises expressed above the line entail the conclusion below the line. To extend the limited axiomatic framework of gFDs and gUCs [25] we use a process similar to that of graph entity dependencies [16]. We derive three new axioms from our definitions of keys and GFDs and establish their soundness and completeness.

- *Reflexivity ($\mathcal{R}$)*: an arbitrary set of attributes A suffices to determine themselves, i.e., for any sets of attributes A, B regardless of the graph pattern, the dependency $AB \mapsto A$ holds true.
- *Extension ($\mathcal{E}$)*: any functional dependency $(Q[\mathcal{X}], A \mapsto B)$ can trivially extend to $A \mapsto AB$ by adding the left hand side of the dependency to the right hand side.
- *Augmentation ($\mathcal{A}$)*: a superkey $(Q[\mathcal{X}], A)$ remains as such by adding any set of attributes A', i.e., $(Q[\mathcal{X}], AA')$ is a superkey.
- *Transitivity ($\mathcal{T}$)*: the combination of two graph functional dependencies $(Q[\mathcal{X}], A \mapsto B)$ and $(Q[\mathcal{X}], B \mapsto C)$ with a shared pattern $Q[\mathcal{X}]$ is a valid functional dependency $(Q[\mathcal{X}], A \mapsto C)$
- *Weakening ($\mathcal{W}$)*: a superkey $(Q[\mathcal{X}], A)$ is equivalently a functional dependency $(Q[\mathcal{X}], A \mapsto \Phi_Q(\mu(\mathcal{X}))$ in which the key determines all the attributes $\Phi_Q(\mu(\mathcal{X}))$ in the pattern.
- *Pullback ($\mathcal{P}$)*: a functional dependency $(Q[\mathcal{X}], A \mapsto B)$ and an associated superkey $(Q[\mathcal{X}], AB)$ entails a key $(Q[\mathcal{X}], A)$ due to the dependency $A \mapsto B$.

The three new axioms refer to changes on the graph pattern.

- *Pattern Extension ($\mathcal{PE}$)*: a functional dependency $(Q[\mathcal{X}], A \mapsto B)$ entails a functional dependency $(Q_e[\mathcal{X}_e], A \mapsto B)$ for any extension with $Q_e[\mathcal{X}_e]$ being a supergraph of $Q[\mathcal{X}]$ ($Q_e[\mathcal{X}_e] \sqsupseteq Q[\mathcal{X}]$).
- *Pattern Augmentation ($\mathcal{PA}$)*: similarly to $\mathcal{PE}$, if $(Q[\mathcal{X}], A)$ is a superkey, then so is $(Q_e[\mathcal{X}_e], A)$, for any $Q_e[\mathcal{X}_e] \sqsupseteq Q[\mathcal{X}]$.
- *Pattern Transitivity ($\mathcal{PT}$)*: functional dependencies $(Q[\mathcal{X}], A \mapsto B)$ and $(Q_o[\mathcal{X}_o], B \mapsto C)$, where the literals in B are common to both graph patterns, i.e., $B \subseteq \Phi_Q(\mu(\mathcal{X})) \cap \Phi_{Q_o}(\mu(\mathcal{X}_o))$, entail $(Q[\mathcal{X}] \sqcup Q_o[\mathcal{X}_o], A \mapsto C)$, where graph pattern $Q[\mathcal{X}] \sqcup Q_o[\mathcal{X}_o]$ is the result of merging $Q[\mathcal{X}]$ and $Q_o[\mathcal{X}_o]$.

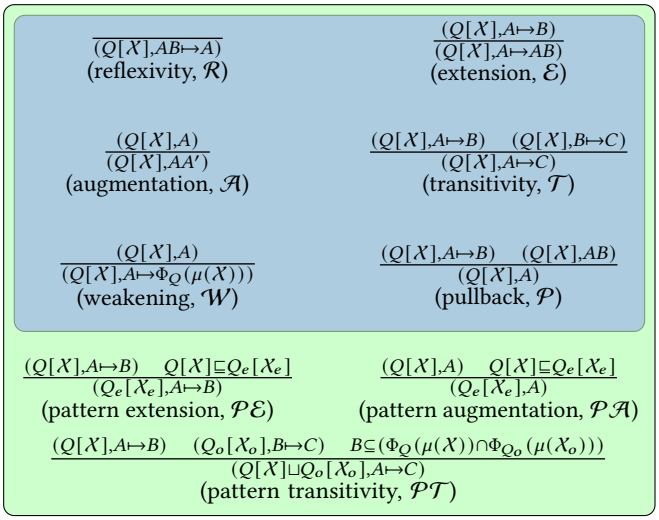

Figure 3: Axiomatization; the first six axioms extend axioms for gFDs and gUCs with single-node patterns [25].

Definition 10 (Closure). *Let Σ be the set of GFDs on a graph G and $\mathbb{R}$ be our extended axiom set $\{\mathcal{R}, \mathcal{E}, \mathcal{A}, \mathcal{T}, \mathcal{W}, \mathcal{P}, \mathcal{PE}, \mathcal{PA}, \mathcal{PT}\}$. Then Σ^+ denotes the set of all GFDs derived from Σ by repeatedly applying the rules of $\mathbb{R}$.*

We extend previous soundness arguments [1, 25] to the three new axioms as follows:

- *Pattern Extension ($\mathcal{PE}$)*: Given that $Q_e[\mathcal{X}_e] \sqsupseteq Q[\mathcal{X}]$, any match of $Q_e[\mathcal{X}_e]$ contains a match of $Q[\mathcal{X}]$. Thus, $Q_e[\mathcal{X}_e]$ reduces the possible matches of $Q[\mathcal{X}]$. Consequently, if $A \mapsto B$ holds in $Q[\mathcal{X}]$, so it does in $Q_e[\mathcal{X}_e]$.
- *Pattern Augmentation ($\mathcal{PA}$)*: Given that $Q_e[\mathcal{X}_e] \sqsupseteq Q[\mathcal{X}]$, any match of $Q_e[\mathcal{X}_e]$ contains a match of $Q[\mathcal{X}]$. Thus, $Q_e[\mathcal{X}_e]$ reduces the possible matches of $Q[\mathcal{X}]$. Therefore, if A forms a key with $Q[\mathcal{X}]$, it does so with $Q_e[\mathcal{X}_e]$ too.
- *Pattern Transitivity ($\mathcal{PT}$)*: Any match h of $Q[\mathcal{X}] \sqcup Q_o[\mathcal{X}_o]$ is decomposable into the component h_Q, which matches $Q[\mathcal{X}]$ and satisfies $A \mapsto B$, and h_{Q_o}, which matches $Q_o[\mathcal{X}_o]$ and satisfies $B \mapsto C$. Since $B \subseteq \Phi_Q(\mu(\mathcal{X})) \cap \Phi_{Q_o}(\mu(\mathcal{X}_o))$, if the variable assignments in two matches assume the same assigned values for literals in A, then they assume the same assigned values for literals in B, and hence the same assigned values for literals in C, ergo $(Q[\mathcal{X}] \sqcup Q_o[\mathcal{X}_o], A \mapsto C)$.

Similarly, we extend completeness arguments as follows:

- *Pattern Extension ($\mathcal{PE}$)*: To obtain $(Q_c(\mathcal{X}_c), A_c \mapsto B_c)$ that follows from Σ, for any premise $(Q_i(\mathcal{X}_i), A_i \mapsto B_i)$ with $Q_c(\mathcal{X}_c) \sqsupseteq Q_i(\mathcal{X}_i)$, we apply $\mathcal{PE}$ to extend the pattern to $Q_c(\mathcal{X}_c)$; we then apply the complete axioms $(\mathcal{R}, \mathcal{E}, \mathcal{A}, \mathcal{T}, \mathcal{W}, \mathcal{P})$ on this fixed pattern.
- *Pattern Augmentation ($\mathcal{PA}$)*: To obtain a key $(Q_c(\mathcal{X}_c), A_c)$ that follows from Σ, we first apply $\mathcal{W}$ on the premise keys $(Q_i(\mathcal{X}_i), A_i)$

to formulate them as functional dependencies, and then apply $\mathcal{PE}$ and other axioms as necessary to derive $(Q_c(\mathcal{X}_c), A_c)$.

- *Pattern Transitivity* ($\mathcal{PT}$): To obtain a $(Q_{c1}(\mathcal{X}_{c1}) \sqcup Q_{c2}(\mathcal{X}_{c2}), A \mapsto C)$ that follows from Σ from derivatives $(Q_i(\mathcal{X}_i), A_i \mapsto B_i)$, we lift any pattern $Q_i(\mathcal{X}_i)$ to the union pattern $Q_{c1}(\mathcal{X}_{c1}) \sqcup Q_{c2}(\mathcal{X}_{c2})$ using $\mathcal{PE}$, and then apply other axioms (especially $\mathcal{T}$) to derive $(Q_{c1}(\mathcal{X}_{c1}) \sqcup Q_{c2}(\mathcal{X}_{c2}), A \mapsto C)$.

Given that all axioms are sound and complete, the syntactic closure for this axiomatization follows.

5 GRAPH NORMAL FORMS

Given the above, we introduce five *graph normal forms (GNFs)*. As in relational databases, these GNFs aim to reduce redundancy and facilitate updates in property graphs, each addressing a concrete cause of redundancy or update anomaly through a set of desirable properties. While the first four normal forms are reminiscent to those in relational databases, the fifth is unique to graph databases.

Applicability. The 1GNF is applicable to most databases as it addresses implicit links and non-atomic attributes, both of which lead to redundancy and update anomalies. The 2GNF addresses attributes describing "implicit entities" within an element. The 3GNF and 3$^+$GNF reduce anomalies caused by the presence of attributes that are replicated since they depend only transitively on the current element's key. The EGNF is the most impactful, it is designed to eliminate any possible redundancy by creating nodes for each single atomic value. These normal forms promote the presence of de-duplicated nodes in place of replicated attribute values. In small-scale databases, splitting data into smaller nodes may be undesirable. Yet, it is possible to enforce normal forms just on specific subgraphs of interest rather than to the entire graph.

5.1 First Graph Normal Form (1GNF)

The First Graph Normal Form (1GNF) ensures that all the attributes of a graph element adhere to the graph model, i.e., no attribute conceals a graph element in disguise. Such attributes would encase complex structures and prevent further normalization. The 1GNF also bars node and edge attributes from serving as standalone foreign keys referencing other nodes or edges, that is, a single attribute cannot constitute an entire foreign key, which would correspond to an edge in the graph representation. Naturally, 1GNF extends relational 1NF. To describe the 1GNF, we define *atomic attributes*.

Definition 11 (Atomic attribute). *An attribute $a \in \Phi(g)$ of graph element $g \in N \cup E$ is atomic if its value $v \in \mathcal{V}$, i.e., its value cannot be decomposed into meaningful graph elements, and is not a foreign key to another element $g' \in N \cup E$.*

This definition extends *atomic* attributes in the relational model to exclude edges that are implicitly modeled as foreign keys of other elements. Moreover, if more complex object types are considered atomic, the normalization can treat such structures as single indivisible values. Using atomic attributes we define 1GNF as follows:

Definition 12 (1GNF). *A graph $G=(N, E, \rho, \ell, \Phi)$ is in **1GNF)** if each of its graph elements contains only atomic attributes, i.e., $\forall g \in N \cup E : \forall a \in \Phi(g)$:a is **atomic**.*

Example 4. *The graph in Figure 2 is not in 1GNF since the attribute paper.keywords contains a list of keywords, each of which may be represented as a node linked to papers via edges. Likewise, edges connecting a paper to an author contain as an attribute the foreign key referring to the affiliation of the author when they wrote the paper. Instead of having foreign keys represented as attributes, the graph in Figure 4 has a mediator node connecting papers, authors, and affiliation, and thus it is in 1GNF.*

5.2 Second Graph Normal Form (2GNF)

We extend the 2NF of relational databases to the case of property graphs via graph functional dependencies. A redundancy arises if a subset of attributes of a graph element functionally depends on another subset of attributes that is not a superkey. Hence, an attribute a of a graph element g should rather be stored on some other element if it fully depends on attributes outside g, i.e., appears on the right-hand side of a functional dependency $(Q[\mathcal{X}], A \mapsto B)$, where g is a valid assignment for $x \in \mathcal{X}$, $A \cap \Phi(x) = \emptyset$, and $B \subseteq \Phi(x)$.

Definition 13 (2GNF). *A graph $G=(N, E, \rho, \ell, \Phi)$ with set of functional dependencies Σ having closure Σ^+ is in 2GNF if: (i) it is in 1GNF, and (ii) $\forall g \in N \cup E$, no attribute $a \in \Phi(g)$ functionally depends either on a proper subset of a candidate key of g or on a key or proper subset of another graph element's candidate key $g' \in N \cup E \wedge g' \neq g$, by any functional dependency in Σ^+.*

Example 5. *Given the example graph in Figure 4a and a GFD $(Q_0[\mathcal{X}], x.name \mapsto y.year)$ using Q_0 in Figure 1, the graph is not in 2GNF since the conference name alone determines the year a paper was published, hence it is redundant to store years in paper nodes. To eliminate this redundancy, we may store the year attribute in the conference node, as in Figure 4b.*

5.3 Third Graph Normal Form (3GNF)

As in relational databases, some redundancies and data anomalies in graphs can arise from dependencies on attributes that are not part of keys, creating transitive dependencies on prime attributes via non-prime attributes. To generalize 3NF to graph structures, we first redefine the notion of prime attribute.

Definition 14 (prime attribute). *An attribute $a \in \Phi(g)$ of a graph element $g \in G$ is **prime** if it is part of a candidate key for g.*

A set of attributes C *transitively depends* on another set A if $(Q[\mathcal{X}], A \mapsto B)$ and $(Q_o[\mathcal{X}_o], B \mapsto C)$ hold, but not $(Q[\mathcal{X}], B \mapsto A)$. The Third Graph Normal Form (3GNF) prevents non-prime attributes from having a transitive dependency on a primary key.

Definition 15 (3GNF). *A graph $G=(N, E, \rho, \ell, \Phi)$ with a set of functional dependencies Σ having closure Σ^+ is in 3GNF if: (i) it is in 2GNF and (ii) $\forall g \in N \cup E$, every non-prime attribute $a \in \Phi(g)$ depends non-transitively on each candidate key $k(g)$, by any functional dependency in Σ^+.*

Example 6. *Consider the graph fragment in Figure 4.b and the graph patterns Q_2 and Q_3 with GFDs FD5 and FD6. Here, the value for a conference publisher transitively depends on the conference name, via the attribute venue. A similar situation holds for the attribute country via city. Thus, the fragment is not in 3GNF. Instead, the fragment in Figure 4c, storing this information in separate nodes, avoids transitive dependencies and satisfies 3GNF.*

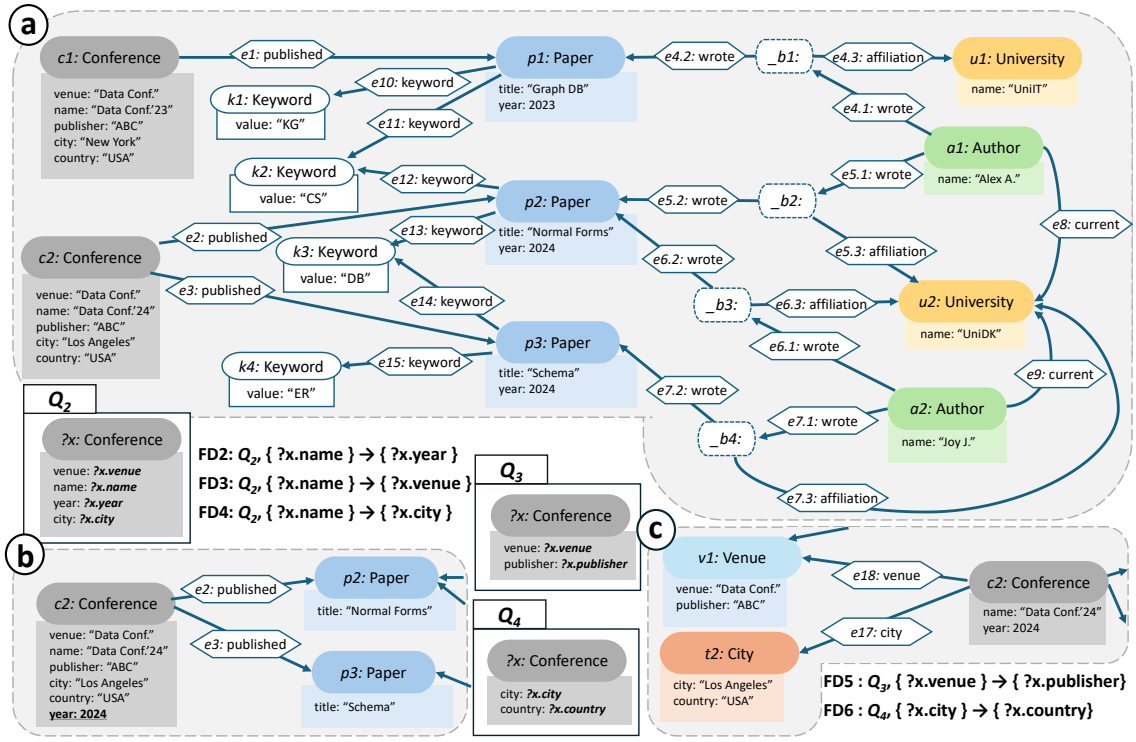

Figure 4: (a) Graph representing the data from Figure 2 while satisfying 1GNF and fragments of graph satisfying (b) 2GNF and (c) 3GNF, with associated patterns and GFDs.

5.4 Restricted Third Graph Normal Form (3^+GNF)

We further restrict 3GNF to obtain an equivalent of BCNF, defining the restricted 3rd Graph Normal Form.

Definition 16 (3^+GNF). *A graph $G=(N, E, \rho, \ell, \Phi)$ with a set of FDs $\mathcal{F}_S$ having closure Σ^+ is in 3^+GNF if: (i) it is in 3GNF and (ii) $\forall g \in N \cup E$ and for each $(Q[\mathcal{X}], A \mapsto B) \in \Sigma^+$, if $A \subseteq \Phi(g)$ then either $B \subseteq A$ or $(Q[\mathcal{X}], A)$ is a superkey for g.*

Theorem 2. *Any graph in 3^+GNF is also in L:P-BCNF [25].*

L:P-BCNF requires for each gFD $\sigma = L{:}P{:}X \rightarrow Y$ with $L \subseteq \mathcal{L}$, $P \subseteq \mathcal{K}$ and $X, Y \subseteq P$ that $Y \subseteq X$ or $L{:}P{:}X$ is a graph-tailored uniqueness constraint [25] for all nodes with labels L and properties P. By Theorem 1, any GFD on a single-node graph pattern is equivalent to a gFD $L{:}P{:}X$, hence for a graph in 3^+GNF, each single-node GFD is equivalent to a gFD satisfying the homologous conditions in Definition 16 or to a homologous uniqueness constraint.

5.5 Extended Graph Normal Form (EGNF)

To remove *all* sorts of data replication in a graph, we should have every node attribute as a candidate key. This condition is more restrictive than 3^+GNF and, while it resembles the Fifth normal form in the relational model [20], it has no direct analogue in relational databases, where it would cause the loss of functional dependencies. Dependency preservation is crucial in the relational model, in the graph data model dependencies can be preserved by edges.

Definition 17 (EGNF). *A graph $G=(N, E, \rho, \ell, \Phi)$ is in **Extended Graph Normal Form (EGNF)** if (i) it is in 3^+GNF and (ii) $\forall g \in N$: $\forall a \in \Phi(g)$: a is a **candidate key** for g.*

Example 7. *In the example of Figure 4c, the value for the attribute country for nodes of type city is replicated across all nodes that represent cities in that country. Similarly, the values for year are replicated across all conferences in a given year. A graph in EGNF, instead, would have country to be its own node and the same for each year.*

As with restrictive normal forms beyond BCNF in the relational model, enforcing the restrictive EGNF causes a proliferation of nodes and therefore increases query complexity.

6 NORMALIZATION ALGORITHMS

Given a graph not satisfying a given normal form, we present normalization algorithms to removing sources of anomalies.

6.1 1GNF Normalization

The transformation of a property graph G to an equivalent G' in 1GNF amends all the cases of non-atomic properties and the use of foreign keys in place of proper edges. We define two helper functions we use in the normalization: (1) decompose(a) that applied on a non-atomic attribute a returns the nested attributes as a list, and (2) getOrCreate(a,G) that either finds and get the element on attribute a if one exists in the graph G or else creates and returns a new element. We propose Algorithm 1 to implement this process. The algorithm first retrieves all edges having some non-atomic attribute a (Line 5). As such, (a) the attribute a is a foreign key

referring to the value of the key attribute of some other node g' (Line 11) or (b) is a composite attribute comprising a list l of attributes (Line 15). In the first case, for the edge e with $\rho(e)=(u,v)$ we create a new blank node v_e. A blank node is a concept borrowed from the RDF data model that refers to a placeholder node without an intrinsic identifier. Although blank nodes are typically not part of property graphs they are an established practice in the modeling of KGs [10]. While they might introduce additional challenges [18], they are in practice auxiliary nodes. The use of blank nodes here mimic the practice of reification and enables relationships among three or more nodes, explicitly prioritizing edges over attributes. Then we create edges (u,v_e), (v_e,v) and (v_e,g') in place of (u,v); if e had some additional attributes excluding a, they are moved to v_e. Similarly, the second case transforms the edge into a new node v_e connected to u and v via (u,v_e), (v_e,v), but also creates a node, connected to v_e, for each attribute in the list of values of a. The algorithm performs a similar verification for the nodes in the graph (Line 23). In the case of a node v with an attribute a being a foreign key to g', the algorithm creates the edge (v,g'). Finally, if a is a list of attributes, we create one node connected to v for each of the attribute values. To preserve the order, we store their position as an attribute on the respective edges.

Efficiency considerations. Algorithm 1 embodies a number of optimizations to avoid unnecessary computations.

First, it prioritizes the detection of violations on edges before validating nodes. This strategy is based on the observation that, during normalization, newly created edges satisfy the 1GNF. Conversely, newly created nodes might still violate the 1GNF due to the presence of multi-layered nested non-atomic properties. Consider a list of lists as an attribute value of a graph element. When this attribute is unrolled for the first time, the newly created nodes still contain a list in violation of 1GNF. Since any form of nested non-atomic properties is split into nodes and new edges are created only as simple links, further checks on them can be disregarded.

Second, when creating a new node, the algorithm checks if a node representing the same entity already exists. If so, it links to the existing node instead of generating a duplicate.

Last, when a (blank) node is created in place of an edge since we are replacing an edge attribute with the role of a foreign key, the nodes are connected to the existing nodes. For instance, in the example for 1GNF in Figure 4 the edge is replaced with a blank node and this is connected to the existing university in addition to the author and papers that were connected by the edge.

6.2 2GNF Normalization

To transform the graph into the 2GNF, we must first identify the graph functional dependencies $\mathcal{F}_S$ that are non-trivial and that are a source of the violations. In particular, the 2GNF disallows any graph element g for which it exist a functional dependency of the form $\mathcal{F}:(Q[\mathcal{X}],A\mapsto B)$ where $A\cap\Phi(g)=\emptyset$ with $B\subseteq\Phi(g)$ or there exists a superkey $(Q[\mathcal{X}],S)\in k(g)$ such that $A\subset S$ (Algorithm 2, Line 2). Then, we isolate the set of functional dependencies where A are attributes of a graph element different than g.

It might happen that two completely different functional dependencies are conditioning the same set of attributes in g. For instance, consider a database in which for students and for workers their

Algorithm 1 Transform to 1GNF

Input: Property Graph $G=(N,E,\rho,\ell,\Phi)$
Output: Property Graph G' satisfying 1GNF
1: Queue $P_e \leftarrow E$; Queue $P_v \leftarrow N$
2: **while** $P_e \neq \emptyset$ **do**
3: $e \leftarrow$ DEQUEUE(P_e)
4: **for** $a \in \Phi(e)$ **do**
5: **if** a not *atomic* **then**
6: $(u,v) \leftarrow \rho(e)$
7: $v_e \leftarrow$ new Node()
8: $\Phi(v_e) \leftarrow \Phi(e) \setminus \{a\}$; $e_s, e_t \leftarrow$ new Edge()
9: $\rho(e_s) \leftarrow (u, v_e)$; $\rho(e_t) \leftarrow (v_e, v)$
10: $N \leftarrow N \cup \{v_e\}$; $E \leftarrow E \cup \{e_s, e_t\}$
11: **if** a is a foreign key **then**
12: $e_a \leftarrow$ new Edge()
13: $v_a \leftarrow$ GETORCREATE(a, G)
14: $\rho(e_a) \leftarrow (v_e, v_a)$; $E \leftarrow E \cup \{e_a\}$
15: **else**
16: **for** $a_i \in$ DECOMPOSE(a) **do**
17: **if** a_i not *atomic* **then**
18: RECDECOMPOSE(a_i)
19: $v_i \leftarrow$ GETORCREATE(a_i, G)
20: $e_{ai} \leftarrow$ new Edge()
21: $\Phi(e_{ai}) \leftarrow \{i\}$; $\rho(e_{ai}) \leftarrow (v_e, v_i)$
22: $E \leftarrow E \cup \{e_{ai}\}$; $N \leftarrow N \cup \{v_i\}$
23: **while** $P_v \neq \emptyset$ **do**
24: $v \leftarrow$ DEQUEUE(P_v)
25: **for** $a \in \Phi(v)$ **do**
// Define the following if-block as recDecompose(a)
26: **if** a not *atomic* **then**
27: **if** a is a foreign key **then**
28: $e_a \leftarrow$ new Edge()
29: $v_a \leftarrow$ GETORCREATE(a, G)
30: $\rho(e_a) \leftarrow (v, v_a)$; $E \leftarrow E \cup \{e_a\}$
31: **else**
32: **for** $a_i \in$ DECOMPOSE(a) **do**
33: **if** a_i not *atomic* **then**
34: RECDECOMPOSE(a_i)
35: $v_i \leftarrow$ GETORCREATE(a_i, G)
36: $e_{ia} \leftarrow$ new Edge()
37: $\Phi(e_{ai}) \leftarrow \{i\}$; $\rho(e_{ai}) \leftarrow (v, v_i)$
38: $N \leftarrow N \cup \{v_i\}$; $E \leftarrow E \cup \{e_a i\}$
39: $\Phi(v) \leftarrow \Phi(v) \setminus \{a\}$
return $G=(N,E,\rho,\ell,\Phi)$

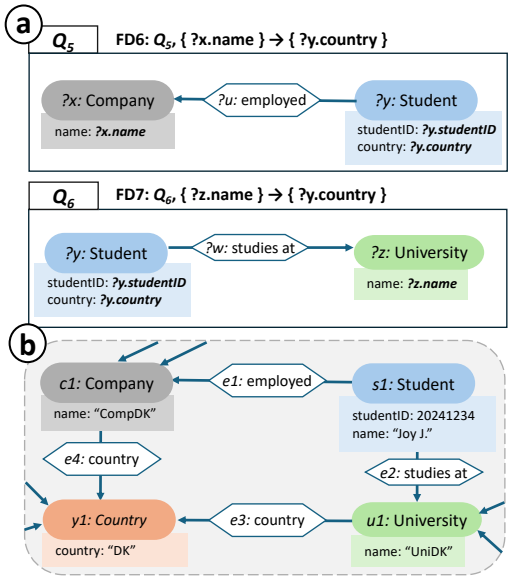

Figure 5: (a) conflicting GFDs and (b) their resolution

university and the company in which they work respectively determine their residing country (as shown in the fragment of Figure 5).

This means that for students that also work both will determine the same value. In this case, the GFDs are said to be conflicting.

Definition 18 (Conflicting dependencies). *Given two GFDs* $\mathcal{F}_1, \mathcal{F}_2 \in \mathcal{F}_S$ *with* $\mathcal{F}_1{:}(Q_1[\mathcal{X}_1], A_1 \mapsto B_1)$ *and* $\mathcal{F}_2{:}(Q_2[\mathcal{X}_2], A_2 \mapsto B_2)$*, they are conflicting with each other if it holds that* $\forall g_1 \in \mu(Q_1)$*,* $\forall g_2 \in \mu(Q_2)$ *with* $g_1 \neq g_2$ *s.t.* $A_1 \subseteq \Phi(g_1)$ *and* $A_2 \subseteq \Phi(g_2)$ *then it holds that* $B_1 = B_2 \wedge A_1 \not\subseteq A_2 \wedge A_2 \not\subseteq A_1$.

Therefore, given two conflicting GFDs $\mathcal{F}_1{:}(Q_1[\mathcal{X}_1], A_1 \mapsto B_1)$ and $\mathcal{F}_2{:}(Q_2[\mathcal{X}_2], A_2 \mapsto B_2)$ and given elements $g_1 \neq g_2 \neq \bar{g}$ s.t. $A_1 \subseteq \Phi(g_1)$, $A_2 \subseteq \Phi(g_2)$, and $B = B_1 = B_2 \subseteq \Phi(\bar{g})$, moving attributes in either graph elements determining the left-hand side of the dependency would not solve the issue.

Algorithm 2 Transform to 2GNF

Input: Property Graph $G = (N, E, \rho, \ell, \Phi)$, $\mathcal{F}_l = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$
Output: Property Graph G' that satisfies the 2GNF

```
1:  L ← ∅; C ← ∅
2:  for F_i : (Q[X], A↦B) ∈ F_l do
3:      if (A∩Φ(x) = ∅ or ∃(Q[X], S) : A⊂S) then
4:          L ← {F_i} ∪ L
5:          if ∃F_j : F_i conflicting with F_i then
6:              C ← C ∪ (F_i, F_j)
7:              L ← L \ {F_i, F_j}
8:  N' ← N; E' ← E
9:  for g ∈ N'∪E' do
10:     for (F_1, F_2) in C do
    // x is the element containing B
11:         if ℓ(x) ⊆ ℓ(g) then
12:             v_g ← getOrCreate(B)
13:             v_1 ← getOrCreate(A_1)
14:             v_2 ← getOrCreate(A_2)
15:             e_1 ← new Edge()
16:             ρ(e_1) ← (v_1, v_g)
17:             e_2 ← new Edge()
18:             ρ(e_2) ← (v_2, v_g)
19:             Φ(g) ← Φ(g) \ (B)
20:             N ← N ∪ {v_g}; E ← E ∪ {e_1, e_2}
21:     for F_i in L do
22:         if ℓ(x) ⊆ ℓ(g) then
23:             if A∩Φ(x) = ∅ then
24:                 y ← getOrCreate(A)
25:                 Φ(y) ← Φ(y) ∪ B
26:                 Φ(g) ← Φ(g) \ B
27:             else
28:                 v_a ← getOrCreate(A ∪ B)
29:                 e_a ← new Edge()
30:                 if g is node then
31:                     ρ(e_a) ← (g, v_a)
32:                     Φ(g) ← Φ(g) \ A ∪ B
33:                 else
34:                     (u, v) ← ρ(g)
35:                     v_e ← getOrCreate(Φ(g) \ (A ∪ B))
36:                     e_s, e_t ← new Edge()
37:                     ρ(e_s) ← (u, v_e)
38:                     ρ(e_t) ← (v_e, v)
39:                     ρ(e_a) ← (v_e, v_a)
40:                     N ← N ∪ {v_e}; E ← E ∪ {e_s, e_t}
41:                 N ← N ∪ {v_a}; E ← E ∪ {e_a}
    return G = (N, E, ρ, ℓ, Φ)
```

Thus, as also illustrated in Algorithm 2 Line 12, we need to create a new graph element g_B in which to move the attributes and then connect it with both g_1 and g_2. This will be the first step of the normalization, to resolve conflicting GFDs.

Finally, we address the case of attributes specifically depending on the attributes from a different element and with of attributes depending on a proper subset of attributes from the primary key of the node in which they reside. In the first case (Line 23), where element A of $\mathcal{F}$ can be derived from a different element from the one on which B resides, we transfer the attributes B to the same element where the attributes of A are located. This is only feasible whenever all attributes of B are derived from a single graph element, this is the reason why conflicting GFDs need to be solved first. In the second case (Line 27), we encounter a functional dependency within a node g that involves part of a candidate key. Here, if A is not the key of the said node, we extract the attributes from the dependency $A \cup B$ to create a new node and connect it with g.

6.3 3GNF and 3⁺GNF Normalization

The normalization in 2GNF ensures that there are no attributes in a graph element that have external dependencies, i.e., non-trivial dependencies from attributes in other graph elements. Therefore, the normalization of a graph that is already in 2GNF into 3GNF follows a process that is similar to that of the third normal form in the relational model as also studied in the context of gFDs and gUC [25]. The process then considers only single node GFDs. Considering the mappings of the graph patterns for those GFDs we obtain a set of relations with the corresponding equivalent relational-form functional dependencies. Then, in the case of 3GNF we employ the normalization process for third normal form and in the case of 3⁺GNF the one for BCNF. In both cases, as in the process for L:P:X [25], we create a new node whenever we split a relation and create edges in place of foreign keys. When we create nodes sharing the same values for the same attributes, we do not duplicate these nodes but rather let edges point to the same node. Further, and unprecedentedly with respect to prior work, we normalize attributes on edges. In case of such attributes, we create an additional new node and add edges to the original source and destination nodes.

Algorithm 3 Transform to EGNF

Input: Property Graph $G = (N, E, \rho, \ell, \Phi)$
Output: Property Graph G' that satisfies the EGNF

```
1:  N' ← N; S_a ← ∅
2:  M_a ← new Map()
3:  for v ∈ N do
4:      for a ∈ Φ(v) do
5:          if {v.a} ∩ M_a[a] ≠ ∅ then
6:              S_a ← S_a ∪ {a}
7:          else
8:              M_a[a] ← M_a[a] ∪ {v.a}
9:  for v ∈ N do
10:     for a ∈ Φ(v) do
11:         if a ∈ S_a then
12:             v_a ← getOrCreate(a)
13:             e_a ← new Edge()
14:             ρ(e_a) ← (v, v_a)
15:             Φ(v) ← Φ(v) \ {a}
16:             N' ← N' ∪ {v_a}; E' ← E' ∪ {e_a}
    return G' = (N', E', ρ, ℓ, Φ)
```

6.4 EGNF normalization

To transform a graph to EGNF we need to go over every node and attribute twice. The first time we fill a map that has as key the attribute type and as value a set of all seen values of the type. While doing so, we check if the current attribute value is already in the set; in that case, we add the attribute type to a list of all types that need to be extracted. Thereafter, we reiterate and check, for all attributes flagged to be extracted, whether a node has already been created for that specific value, and, if not, create it. Then we add an edge connecting the original node and the attribute value node. If a graph is not in EGNF, we decompose all duplicated elements of superkeys

into new nodes. Nodes with no attributes (like relationship nodes created for 1GNF) utilize the pattern key from the graph structure.

6.5 Correctness and Complexity

We substantiate our normalization algorithms with proofs of correctness and complexity analysis. We present a proof for Algorithm 1; a similar argument holds for the other normalization algorithms. Besides, the normalization transformations we perform are lossless, therefore we can transform any query over the unnormalized data to its equivalent over the normalized data by modifying graph patterns following the normalizing transformation.

Proposition 1. *Algorithm 1 transforms a graph G in 1GNF.*

Proof. (Sketch): Given a property graph $G = (N, E, \rho, \ell, \Phi)$ we distinguish two cases. (1) The graph is in 1GNF . As such, all the attributes are atomic and the algorithm does not change the graph. (2) The graph is not in 1GNF . *For edges*, the algorithm replaces the edge with a new node v_e, connecting the original source and target nodes with new edges. For foreign keys, an additional node containing the composite attributes is added to v_e. Otherwise, each value is assigned a newly created node, which is linked to v_e. All newly created edges and nodes lack properties, ensuring compliance with 1GNF. If a node contains a foreign key, the algorithm replaces the key with an edge. *For nodes requiring decomposition*, new nodes are created and connected to the currently evaluated node, and these new nodes are recursively decomposed if necessary. After resolving violations in graph every element, the graph is in 1GNF. □

Transformations across all GNFs introduce additional nodes and edges, making the property graph more explicit: implicit connections are removed, and redundancies captured by GFDs are reassigned to existing or newly created nodes. The EGNF applies the most aggressive transformation. Therefore, its application should be considered depending on the specific needs and desired outcome.

Time complexity. Algorithm 1 inspects all attributes for all edges and nodes to identify foreign keys, incurring a time complexity of $O(|N||\mathcal{R}|+|E||\mathcal{R}|)$. For 1GNF, Algorithm 1 inspects the attributes of all nodes and edges to decompose non-atomic attributes and resolves violations recursively on each nested attribute. Let m_a be the maximum number of attributes on any element $g \in G$, and m_l be the maximum nesting level. The algorithm takes $O(|E|m_a m_l)$ to check the edges and $O((|N|)m_a m_l)$ for the nodes, incurring a time complexity of $O((|N|+|E|)m_a m_l)$. Algorithm 2 inspects each graph element and each functional dependency in $\mathcal{F}_l$ once, ensuing a complexity of $O((|N|+|E|)|\mathcal{F}_l|)$. For EGNF, Algorithm 3 inspects each node and attribute once, running in $O(|N|m_a)$.

7 EXPERIMENTS

We show that our normal forms effectively reduce redundancies in practice while containing data size increase and insertion times. We implement the algorithms in Section 2.2 in C++ and conduct experiments on a MacBook Pro 2021 with a M1 Pro chip and 32GB RAM. We release the data and open-source the code[3]. We compile with clang v14.0.0 using flag `-std=c++11` and `-O2` optimization.

Datasets. We select six datasets (described in Table 2), the two larger ones, DBLP and DBpedia, show how a real life dataset would change during normalization substantiating the applicability of our proposal, also extending prior work [25] with 4 more datasets.

Table 2: Dataset characteristics; $\uplus$ is the multiset (bag) union.

Dataset	Nodes ($\|N\|$)	Edges ($\|E\|$)	Attribute values ($\|\uplus\mathcal{R}\|$)
Northwind	1 035	3 139	15 159
Offshore	2 016 523	2 901 722	23 500 561
DBLP	10 584 818	23 084 323	158 724 253
DBpedia	19 864 182	45 414 669	113 213 308
MIMIC-III	251 442	5 203 106	2 292 326
GO	51 693	95 609	149 455

- **Northwind**[4], a synthetic relational database transformed into a property graph for testing Neo4j; it contains data for a fictitious import and export company that trades food across the world.
- **Offshore**[5], released by the International Consortium of Investigative Journalists (icij) to model data about ~2M entities and companies of offshore structures created to avoid taxes.
- **DBLP**[6], a database describing ~10.6M authors, papers, affiliations and citations; we downloaded version 1.3.2024, encoded in RDF format and mapped to an equivalent PG.
- **DBpedia**[7] [9], a large KG built extracting information from Wikipedia on domains such as music, biology, and space; we downloaded the English version from the snapshot of 01.10.2020.
- **MIMIC-III**[8] [19], a real-world medical dataset from an intensive care unit containing anonymized patient records featuring treatments, mortality, and other clinical information.
- **GO**[9] [12], a biological dataset that describes genes across species; each term in the ontology may have multiple parents.

This dataset selection covers domain-specific (Offshore, DBLP, GO), relational-to-graph (Northwind, MIMIC-III), and general (DBpedia) datasets. We use Northwind and Offshore for the sake of comparability to previous work [25]. We include DBLP, a graph with a few node types (e.g., authors, publications), and DBpedia, which has over 320 node types [23], to study how our GNFs apply to diverse property graphs. With the inclusion of a medical relational dataset (MIMIC-III) and a biological system ontology (GO), we showcase applications to specialized scientific domains.

Performance measures. We measure redundancy and size:

- **Redundancy:** When we create a new node to represent an attribute value, all instances containing that attribute value connect to this new node, avoiding replication. We monitor the number of attribute values when applying each normal form as a measure of redundancy reduction, hence normalization effectiveness.
- **Size:** We measure the size of the dataset after every normalization step in number of edges, nodes, and disk space consumption.

7.1 Redundancy reduction

We compare our normalization (as in Section 5) to L:P-BCNF [25], which is subsumed by our 3^+GNF (cf. Theorem 2) We present results in bar charts and measurements for L:P-BCNF with horizontal lines.

[3] https://github.com/AU-DIS/GraphNormalization

[4] https://github.com/neo4j-graph-examples/northwind
[5] https://offshoreleaks.icij.org/pages/database
[6] https://blog.dblp.org/2022/03/02/dblp-in-rdf/
[7] https://www.dbpedia.org/
[8] https://physionet.org/content/mimiciii/1.4/
[9] https://www.geneontology.org/

Table 3: Functional dependencies ($Q[X], A \mapsto B$) enforced; 'mined' denotes GFDs found via explorative search.

Dataset	GNF	Source	Pattern ($Q[X]$)	FD
Northwind	2GNF	[25]	$(c:Customer) \rightarrow (o:Order)$	$\{c.customerID\} \mapsto \{o.shipCity, o.shipName, o.shipPostalCode, o.shipCountry, o.shipAddress, o.shipRegion\}$
	3GNF	mined	$(c:Customer)$	$\{c.city\} \mapsto \{c.shipCity, c.shipCountry, c.country\}$
	3GNF	mined	$(p:Product)$	$\{p.unitPrice, p.unitsInStock\} \mapsto \{p.quantityPerUnit, p.unitsOnOrder, p.reorderLevel, p.discontinued\}$
Offshore	2GNF	mined	$(i:Intermediary) \rightarrow (e:Entity)$	$\{i\} \mapsto \{e.sourceID\}$
	3GNF	[25]	$(e:Entity)$	$\{e.service\_provider\} \mapsto \{e.valid\_until, e.scourceID\}$
	3GNF	mined	$(e:Entity)$	$\{e.closed\_date, e.incorporation\_date\} \mapsto \{e.registration\_date\}$
	3GNF	mined	$(e:Entity)$	$\{e.ibcRUC, e.struck\_off, e.lastEditTimestamp\} \mapsto \{e.inactivation\_date\}$
	3GNF	mined	$(e:Entity)$	$\{e.country\_codes\} \mapsto \{e.company\_type\}$
DBLP	2GNF	mined	$(a:Article) \rightarrow (j:JournalVolume)$	$\{j.name\} \mapsto \{a.publishedInJournalVolume\}$
	2GNF	mined	$(p:Publication) \rightarrow (g:group)$	$\{p.isbn, p.doi\} \mapsto \{g.primaryAffiliation\}$
	2GNF	mined	$(p:Publication) \rightarrow (g:group)$	$\{p.publishedAsPartOf, p.publishedInJournal\} \mapsto \{g.webpage\}$
	2GNF	mined	$(i:Inproceedings) \rightarrow (s:SeriesVolume) \mapsto \{s.name\}$	$\{i.yearOfPublication, i.publishedInSeriesVolume\}$
	3GNF	mined	$(p:Person)$	$\{p.primaryHomepage, p.creatorName\} \mapsto \{p.homepage\}$
	3GNF	mined	$(p:Publication)$	$\{p.publishedInSeries\} \mapsto \{p.publishedIn\}$
	3GNF	mined	$(p:Publication)$	$\{p.publishedBy\} \mapsto \{p.publishersAddress\}$
	3GNF	mined	$(p:Publication)$	$\{p.listedOnTocPage\} \mapsto \{p.publishedAsPartOf, p.publishedInBook, p.yearOfEvent\}$
	3GNF	mined	$(p.Publication)$	$\{p.pagination, p.documentPage\} \mapsto \{p.publishedInBookChapter, p.primaryDocumentPage, p.monthOfPublication\}$
DBpedia	2GNF	mined	$(b:Bio-Classification) \rightarrow (c:Bio-Categories)$	$\{b.taxon\} \mapsto \{c.regnum\}$
	2GNF	mined	$(t:Team) \rightarrow (p.President)$	$\{p.name\} \mapsto \{t.finalsSeries\}$
	3GNF	mined	$(s:Settlement)$	$\{s.pushpinMap\} \mapsto \{s.translitLang\}$
	3GNF	mined	$(m:MusicGenre)$	$\{m.filename\} \mapsto \{m.description\}$
	3GNF	mined	$(m:MusicGenre)$	$\{m.quote, m.culturalOrigins, m.stylisticOrigins, m.regionalScenes\} \mapsto \{m.source\}$
GO	3GNF	mined	$(n:\_)$	$\{n.creation\_date\} \mapsto \{n.created\_by\}$
MIMIC-III	3GNF	mined	$(a:Admission)$	$\{a.discharge\_location\} \mapsto \{a.hospital\_expire\_flag\}$
	3GNF	mined	$(p:Patients)$	$\{p.DOD\} \mapsto \{p.expire\_flag\}$

Used GFDs. As input to the algorithm, we use the GFDs listed in Table 3, which we extracted by mining them the respective datasets, except for two ones, which we retrieved that we from previous work's codebase as indicated.

Number of attributes. Figure 7 shows the reduction in the number of attribute values after normalization. All datasets present a significant reduction. However, the effect of normalization varies among datasets. Northwind, a relational database converted into a graph, has only atomic attributes. Thus, normalization leaves it unaltered (None=1GNF). In 2GNF, we apply the set of functional dependencies from Table 3. 3GNF enforces the second functional dependency in Table 3, hence moves the shipping address from *order* to *customer* nodes. 3^+GNF does not detect additional violations, while EGNF requires each attribute to be a candidate key.

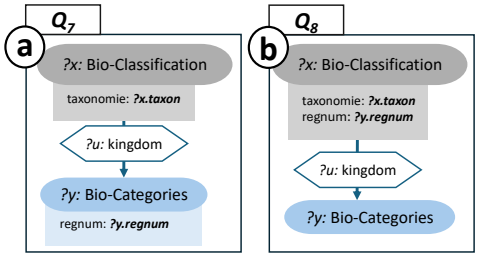

Figure 6: 2GNF example normalization on DBpedia.

On DBLP and DBpedia, 1GNF normalization incurs significant redundancy reduction. On DBLP, attributes *affiliation* and *published in* are lists, while DBpedia is a massive dataset with repeated attribute values. On DBLP, 2GNF consolidates publication information from journals or series into a high-level volume that encompasses them, reducing approximately 2000 attribute values. Ultimately, it ensures each attribute is a candidate key for its node, yielding smaller reduction than previous steps and EGNF. On DBpedia, 2GNF enforces the dependency in Figure 6(a) on the biology classification system to obtain the pattern in Figure 6(b), which moves *regnum* to the *bio-classification* node. 3GNF uses a dependency on *Settlement* nodes. Lastly, EGNF further reduces attribute values.

As the Offshore, GO and MIMIC-III datasets are already well-structured in their given form. To apply 3GNF to Offshore, we consider the functional dependencies in Table 3, removing 1 515 805 copies of attribute values. The EGNF reduces the duplicate values, resulting in reduction by 45%. Similarly, in GO we bring about a reduction of about 50%, and in MIMIC-III about 80% with EGNF.

On Northwind and Offshore, we present the results of normalization by *graph-tailored functional dependencies* [25] with the red line in Figure 7, which our 3GNF clearly surpasses.

Graph size. The increase in the number of nodes and edges resulting from each normalization (Figure 7) corresponds to the reduction in replicated attribute values. Notably, 1GNF, 2GNF, 3GNF, and 3^+GNF do not introduce significant overhead. In contrast, EGNF generates a node for each non-key attribute. Hence, this normalization level should be employed carefully for cases where few large data values are replicated across multiple nodes and thus their storage and update cost is offset by edge cost. An exception occurs in DBpedia, where 1GNF normalization significantly impacts the structure by identifying numerous composite attributes. This experiment underscores the critical role of normalization in preventing data redundancy. Table 4 highlights that the relative nodes and edge growth compared to the original graph is highly data dependent.

Table 4: Growth of normalized graphs.

	Growth Factor									
	Nodes					Edges				
Dataset	1GNF	2GNF	3GNF	3^+GNF	EGNF	1GNF	2GNF	3GNF	3^+GNF	EGNF
Northwind	1	1	1.17	1.17	2.61	1	1	2.61	2.61	3.15
Offshore	1	1	1.12	1.12	2.79	1	1	1.35	1.35	6.45
DBLP	2.29	2.29	2.34	2.34	4.19	2.84	2.84	3.16	3.16	6.53
DBpedia	1.45	1.45	1.45	1.45	1.92	2.69	2.69	2.69	2.69	3.08
GO	1.02	1.02	1.38	1.38	1.42	1.06	1.06	1.26	1.26	1.51
MIMIC-III	1	1	1.05	1.05	2.42	1	1	1.03	1.03	1.37

Space consumption. Lastly, Figure 7 presents the storage footprint required for normalization, obtained by measuring the data size exported to JSON files compatible with Neo4j (https://neo4j.com). As the storage consumption of normal forms varies according to the attribute values and their string sizes, their memory footprint

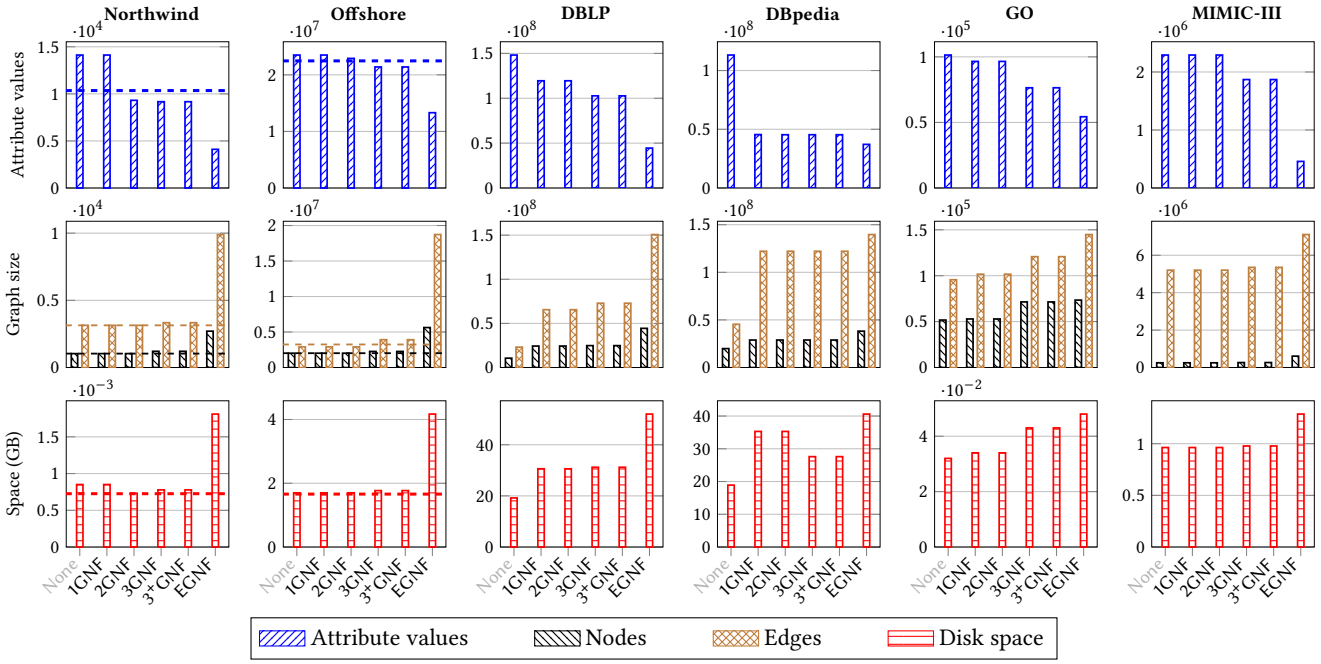

Figure 7: Redundancy reduction in number of attribute values (top); graph size in number of nodes and edges (middle); disk space consumption (bottom) before (None) and after each normalization; dashed lines show results by L:P-BCNF [25].

can drop even as the graph size grows. Consistently, EGNF requires more storage space than others. This highlights that EGNF is an extreme option, used only when necessary; as with relational normal forms, one can stop at the level that best suits the application.

7.2 Query time

Table 5 present the execution times for queries of three types (search, count, update) on the unnormalized and EGNF data, obtained by executing 10 queries of each type in Neo4j. We created the queries automatically for the unnormalized data and transformed each query to a semantically equivalent query on normalized data by applying the normalizing transformations on it.

Table 5: Query time *before* and *after* normalization.

	Query Execution Time (ms)					
	Original			Normalized		
Dataset	search	count	update	search	count	update
Northwind	18.9 ± 4.1	16 ± 1.5	21 ± 2.8	18.4 ± 4.6	16.8 ± 3.2	22.3 ± 2.8
Offshore	28.2 ± 5.5	28.2 ± 5.6	28.5 ± 3.8	25.9 ± 3	26.4 ± 4	28.4 ± 4.2
DBLP	22.2 ± 3.8	20.9 ± 5.1	23.8 ± 2.7	26 ± 4.5	32.2 ± 9.6	47 ± 37
DBpedia	21.1 ± 1.4	21 ± 1.7	26.1 ± 1.3	28 ± 3.8	29 ± 3.7	56.1 ± 4.9
GO	37.9 ± 13.3	26.5 ± 5.6	35.5 ± 3.2	28 ± 5.1	26.1 ± 2.5	62.1 ± 48.1
MIMIC-III	33 ± 13.7	22.7 ± 1.1	31.4 ± 4.5	24.4 ± 2.1	25.1 ± 3.7	47.7±13.4

A *search* query returns the identifiers of nodes that match the search criterion:

```
MATCH (n:NODE) WHERE n.{prop}="{value}" AND n.'~label'="{label}"
RETURN n.IRI
```

Their transformation by EGNF varies based on the attribute selected in the transform, therefore some queries need no change, while others require a path from an original node n to the post-normalization location of its associated information. For all datasets, the differences in execution time fall within the standard deviation.

A *count* query returns the number of occurrences of a value:

```
MATCH (n:NODE) WHERE n.{prop}="{value}" AND n.'~label'="{label}"
RETURN count(n)
```

The associated transformation is similar to the one for search queries. The execution time remains similar, though usually a few milliseconds higher on the normalized graph.

Lastly, an *update* query changes a value in the dataset:

```
MATCH (n:NODE)
WHERE n.{prop} = "{value}" AND n.'~label' = "{label}"
SET n.{prop} = "{new_value}"
RETURN count(n.{prop})
```

To transform these queries, we add a new node and an edge directed to it. Execution times are therefore perceptibly higher in this case for all datasets; that is the price one pays for the normalizing transformation. Still, even when the runtime doubles, as in DBpedia, it remains 62.1ms on average.

8 CONCLUSION

We introduced a comprehensive proposal for the normalization of graph databases using graph functional dependencies. As previous attempts had overlooked dependencies extending to patterns across edges, we proposed five normal forms, each building on its predecessor, that attend to cross-edge dependencies to reduce redundancies. Our proposals are logically aligned with normal forms for relational databases and supersede prior efforts on graph normalization. We provided a detailed theoretical analysis and an experimental study with real-world data to illustrate that the normalization we propose efficaciously reduces stored attribute values without information loss and without incurring a significant query execution time overhead. In the future, we aim to consider cyclic patterns and other ways to resolve conflicting dependencies.

REFERENCES

[1] 1974. Dependency structures of data base relationships. In *Information Processing 74*. North-Holland, 580–583.

[2] 2024. Information technology – Database languages – GQL. https://www.iso.org/standard/76120.html

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[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 Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2423–2436. https://doi.org/10.1145/3448016.3457561

[5] Renzo Angles, Aidan Hogan, Ora Lassila, Carlos Rojas, Daniel Schwabe, Pedro Szekely, and Domagoj Vrgoč. 2022. Multilayer graphs: a unified data model for graph databases. In *5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. https://doi.org/10.1145/3534540.3534696

[6] Marcelo Arenas. 2006. Normalization theory for XML. *ACM SIGMoD Record* 35, 4 (2006), 57–64.

[7] Marcelo Arenas and Leonid Libkin. 2004. A normal form for XML documents. *ACM Transactions on Database Systems (TODS)* 29, 1 (2004), 195–232.

[8] Marcelo Arenas and Leonid Libkin. 2005. An information-theoretic approach to normal forms for relational and XML data. *Journal of the ACM (JACM)* 52, 2 (2005), 246–283.

[9] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*. Springer, 722–735.

[10] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *Comput. Surveys* 56, 2 (2023), 1–40.

[11] Edgar F Codd. 1972. Further normalization of the data base relational model. *Data base systems* 6 (1972), 33–64.

[12] Gene Ontology Consortium. 2004. The Gene Ontology (GO) database and informatics resource. *Nucleic acids research* 32, suppl_1 (2004), D258–D261.

[13] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for graphs. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1590–1601.

[14] Wenfei Fan, Wenzhi Fu, Ruochun Jin, Ping Lu, and Chao Tian. 2022. Discovering association rules from big graphs. *Proceedings of the VLDB Endowment* 15, 7 (March 2022), 1479–1492. https://doi.org/10.14778/3523210.3523224

[15] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Capturing associations in graphs. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 1863–1876. https://doi.org/10.14778/3407790.3407795

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

[17] Claudio Gutierrez and Juan F. Sequeda. 2021. Knowledge graphs. *Commun. ACM* 64, 3 (feb 2021), 96–104. https://doi.org/10.1145/3418294

[18] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. 2014. Everything you always wanted to know about blank nodes. *Journal of Web Semantics* 27 (2014), 42–69.

[19] Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific data* 3, 1 (2016), 1–9.

[20] William Kent. 1983. A simple guide to five normal forms in relational database theory. *Commun. ACM* 26, 2 (1983), 120–125.

[21] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes constraint language (SHACL). *W3C Candidate Recommendation* 11, 8 (2017), 1.

[22] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos Manuel Lopez Enriquez, Ronak Sharda, et al. 2022. The OneGraph vision: Challenges of breaking the graph model lock-in. *Semantic Web* 14, 1 (2022), 125–134.

[23] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web* 6, 2 (2015), 167–195.

[24] Philipp Skavantzos, Uwe Leck, Kaiqi Zhao, and Sebastian Link. 2023. Uniqueness Constraints for Object Stores. *ACM J. Data Inf. Qual.* 15, 2 (2023), 13:1–13:29. https://doi.org/10.1145/3581758

[25] Philipp Skavantzos and Sebastian Link. 2023. Normalizing Property Graphs. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 3031–3043. https://doi.org/10.14778/3611479.3611506

[26] Philipp Skavantzos, Kaiqi Zhao, and Sebastian Link. 2021. Uniqueness constraints on property graphs. In *International Conference on Advanced Information Systems Engineering*. Springer, 280–295.

[27] Zahir Tari, John Stokes, and Stefano Spaccapietra. 1997. Object Normal Forms and Dependency Constraints for Object-Oriented Schemata. *ACM Trans. Database Syst.* 22, 4 (1997), 513–569. https://doi.org/10.1145/278245.278247

[28] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An open-source graph database system. *Data Intelligence* 5, 3 (2023), 560–610.