Computing Why-Provenance for Property Graph Queries

Koumudi Ganepola
TU Wien
Vienna, Austria
koumudi.ganepola@tuwien.ac.at

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

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

ABSTRACT

Labeled Property Graphs (LPGs) have gained widespread adoption, supported by the ongoing standardization of the Graph Query Language (GQL). As LPGs are frequently constructed through the integration of heterogeneous data sources, inconsistencies and errors may propagate into query results. To explain and validate such results, it is essential to determine which parts of the input graph contributed to a given answer – an aspect captured by data provenance. In this paper, we present a formal definition of why-provenance tailored specifically to GQL, characterizing the fine-grained subgraph that contribute to a query answer. Building on the Core-GQL algebraic foundation, we further propose an efficient query-rewriting-based algorithm for computing provenance, leveraging the native execution capabilities of existing GQL-compliant graph database systems. Our extensive experimental evaluation demonstrates the feasibility and scalability of our approach, showing that provenance can be computed with moderate overhead that scales effectively with graph size.

PVLDB Reference Format:
Koumudi Ganepola, Maxime Jakubowski, and Katja Hose. Computing Why-Provenance for Property Graph Queries. PVLDB, 19(11): 3552 - 3564, 2026.
doi:10.14778/3836663.3836708

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dmki-tuwien/gql-provenance.

1 INTRODUCTION

The property graph model is widely adopted for its intuitive and flexible approach to graph data modeling, representing real-world entities and relationships as nodes and edges, with labels and properties (key-value pairs) capturing their types and attributes [2]. This growing adoption of property graphs has led to the standardization of the *G*raph *Q*uery *L*anguage (GQL) [11], providing a formal foundation for graph querying. In turn, this standardization has fostered extensive research, ranging from theoretical foundations [8, 10, 14] to the development of practical systems [9] in recent years.

In many real-world applications, property graphs are constructed by integrating data from heterogeneous sources, such as relational databases, semi-structured data, and external services, into a unified graph representation. While this integration enables rich cross-domain analysis, it can propagate data quality issues into the graph, thereby increasing the risk of unexpected query results. Figure 1 shows a property graph G_{crypto} of a cryptocurrency network that captures information about wallets, their owners, and the transfers between them. This graph is constructed by integrating the wallet information from the blockchain ledger, owner data from a relational database, risk scores of wallet holders from external services, and transfer data extracted from logs.

High-value transfers in a cryptocurrency network are often considered risk indicators. Consider the following property graph query (GQL query), which identifies the owners of wallets that initiate suspiciously high-value transfers and retrieves their names along with associated risk scores:

Q_{transf} :
```
MATCH (p:Person)-[:Owns]->(w1:Wallet)-[t:Transfer]->(w2)
WHERE t.amount > 4000
RETURN p.name AS name, p.riskScore AS riskScore
```

The evaluation of Q_{transf} on graph G_{crypto} yields the following result, where each tuple has an identifier μ_i:

$[\![Q_{transf}]\!]_{G_{crypto}}$ =

	name	riskScore
μ_1 :	"Alice"	0.7
μ_2 :	"Bob"	0.4

The presence of tuple μ_2 in the result is surprising because, given Bob's low risk score, he would normally not be flagged for suspicious activity. This anomaly raises further investigative questions that can be addressed using provenance.

Why-provenance can answer the question *why does Bob appear in the result?*; as provenance for tuple μ_2 we would obtain the set $\{\{n_2, n_5, n_4, e_2, e_6, l_2, l_9, l_5, l_{13}, k_4, k_6, k_{19}\}\}$, where each n_i, e_j, l_r, and k_s is a unique identifier of a node, edge, label, or property from graph G_{crypto} (Figure 1). Why-provenance represents all derivations of a query result as a set of sets, with each nested set listing the data items contributing to a particular derivation [5, 6]. By making explicit which data items of a graph are responsible for the result, why-provenance enables debugging, view maintenance, and internal error tracking and, when combined with source annotations, supports tracing errors back to external data sources [15, 20, 23]. In the property graphs, capturing provenance at a fine-grained level is crucial to keep these tasks tractable, as nodes and edges can carry many properties and labels irrelevant to the query result.

Provenance has been extensively studied in the context of relational queries [6, 12, 18, 27] and has also been explored in the contexts of SPARQL [4, 13, 19] and regular path queries (RPQs) [17, 26]. While these developments are highly relevant to GQL, which comprises of two primary components, pattern matching and linear composition of relational operators, defining provenance models for GQL that can capture the relevant finer details, i.e., labels and properties, is not straightforward.

In this paper, we present the first comprehensive formalization of provenance for property graph queries, capturing not only nodes

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836708

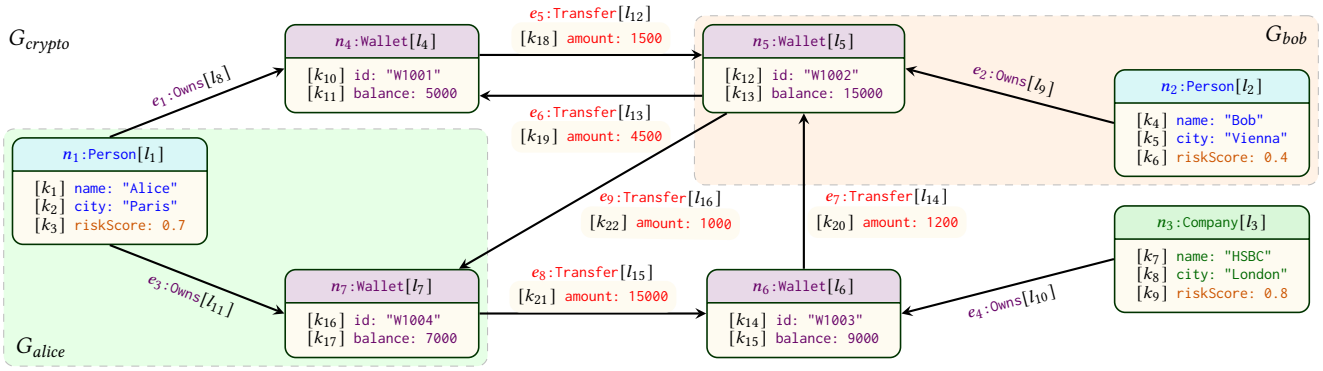


Figure 1: A graph of a cryptocurrency network (G_{crypto}). Nodes of the same color originate from the same source. In addition to the node and edge identifiers, each label and property is also annotated with a unique identifier.

and edges but also their associated properties and labels at a fine-grained level. Our work supports queries ranging from simple path patterns to complex patterns, including RPQs. Additionally, we propose a novel query-rewriting-based approach for computing provenance annotations that is applicable to any GQL-compliant graph database system. In summary, this paper makes the following contributions: (i) define *why-provenance* for property graph queries; (ii) propose a generalized approach for computing provenance annotations; and (iii) conduct an extensive experimental evaluation of the runtime using Neo4j.

The remainder of this paper is structured as follows. Section 2 reviews related work in computing provenance and Section 3 introduces the preliminaries required for our approach. While Section 4 defines why-provenance for property graph queries, Section 5 presents how to compute provenance. Section 6 then discusses our evaluation. Finally, we conclude the paper and outline directions for future work in Section 7.

2 RELATED WORK

Provenance for relational queries: The notions of *why-provenance* and *where-provenance* were introduced by Buneman et al. [5] in the context of a deterministic semi-structured data model and an associated query language. Cheney et al. [6] later reformulated these notions for the relational setting, explicitly characterizing them for relational algebra restricted to select–project–join–union (SPJU) queries. Broadening the study of data provenance, Green et al. [18] and Geerts et al. [12] proposed the notion of *how-provenance* using provenance semirings, in which provenance annotations are represented as polynomials, providing an algebraic characterization of data derivations.

Numerous approaches have since been proposed to realize provenance in practice through query rewriting techniques. These techniques instrument queries with provenance annotations while leveraging the database system's native optimizations and execution capabilities. Glavic et al. [16] introduced Perm, an approach that integrates provenance support directly into the query engine by rewriting relational operators to propagate provenance annotations. Unlike Perm, ProvSQL [27] implements provenance as a PostgreSQL extension that instruments queries within the query engine. In contrast, approaches, such as GProM [3], adopt a middleware-based approach by rewriting queries externally without modifying the underlying database engine. These middleware-based rewriting approaches offer flexibility in supporting diverse annotation schemes while preserving compatibility with existing database systems.

However, existing approaches for computing provenance in relational database systems are not directly applicable to property graph queries. The absence of a predefined schema in property graphs, combined with the lack of a notion of paths and support for pattern matching in existing relational provenance computation approaches, limits their applicability to this setting.

Provenance for graph queries: Damásio et al. [7] and Geerts et al. [13] investigated how-provenance for RDF and SPARQL by computing provenance annotations through the translation of SPARQL queries into relational queries. Building on this foundation, Wylot et al. [29] introduced TripleProv, a query-rewriting based approach that relies on a specialized engine that stores data as RDF molecules, which restricts its applicability on existing engines. In contrast, SPARQLProv [19] and NPCS [4] are query-rewriting based approaches that can work on existing SPARQL engines, supporting both monotonic and non-monotonic queries. Although these approaches also work abstractly on a graph data model, they are not immediately applicable to property graphs. Differences in data models (due to the presence of properties and labels in property graphs) and query semantics, prevent the direct applicability of these SPARQL-based approaches to property graphs queries. Furthermore, the related work on SPARQL focuses on other technical challenges: in RDF, one needs to account for more complex annotation schemes ("reification schemes") for retrieving triple IDs, which do not naturally occur in RDF datasets. In contrast, as we shall see later, in property graphs these are almost naturally given.

Ramusat et al. [26] focused on computing graph-based algorithms using application-specific semiring provenance techniques, showing that the most effective algorithm depends on the semiring used. Following this work, Grahne et al. [17] studied universal provenance, the most general form of provenance for RPQs. As expected, both of these studies base their approaches on the paths corresponding to answers to an RPQs.

However, to the best of our knowledge, no existing work studies provenance for property graph query results. While provenance for relational queries and for RPQs has been explored, it remains unclear how these can be combined, and how provenance should be captured for GQL pattern matching, which extends beyond the expressiveness of RPQs [10, 22]. Furthermore, capturing provenance for labels and properties adds additional complexity. Our work addresses this gap by formalizing provenance for property graphs and proposing a query rewriting approach for its computation, providing a foundation for tracing, auditing, and reasoning about the derivation of property graph query results, including those involving complex graph traversals.

3 PRELIMINARIES

We define the fundamental notions of *property graphs*, and the associated standard *query language* GQL.

3.1 Property Graphs

Since the foundations of provenance and graph queries are rooted in the relational model, we first define the necessary relational concepts and their connection to LPGs.

Relational Model. We use the standard definition of the relational model [1]. Assume three countable, mutually disjoint sets of *attributes* $\mathbf{A}$, *relation names* $\mathbf{N}$, and *domain elements* $\mathbf{dom}$ including the distinguished element null. We associate every relation name $R \in \mathbf{N}$ with a finite set of attributes $U \subseteq \mathbf{A}$, denoted by $attr(R) = U$. A *relation schema* is simply a relation name R. Additionally, when $attr(R) = U$ we write $R[U]$ to make the set of attributes explicit. A *database schema* $\mathbf{D}$ is a set of relation schemas. A tuple is a function $\mu : \mathbf{A} \mapsto \mathbf{dom}$. We write $attr(\mu)$ to denote the attributes U on which μ is defined. A *null-tuple* μ_U^{null} maps every attribute from U to null. An instance D of a database schema $\mathbf{D}$ is a partial function mapping relation schemas $R[U] \in \mathbf{D}$ to sets of tuples, denoted by $[\![R]\!]_D$, such that for every $\mu \in [\![R]\!]_D$ we have $attr(\mu) = U$. We call $[\![R]\!]_D$ a *relation*. For convenience of notation, we often represent database instances as sets of *atoms*: for a relation schema R we assume an order on the elements of $attr(R) = [A_1, \ldots, A_n]$, such that any tuple μ for $attr(R)$ can be written as the atom $R(a_1, \ldots, a_n)$ such that for $1 \leq i \leq n : \mu(A_i) = a_i$.

Example 3.1. Consider the database schema consisting of relation schemas $\mathsf{Transfer}[tid, src, tgt]$ and $\mathsf{Amount}[tid, amount]$. The instance consists of tuple $\mu_1 := \{tid \mapsto t_4; src \mapsto w_4; tgt \mapsto w_3\}$ for Transfer and $\mu_2 := \{tid \mapsto t_4; amount \mapsto 15000\}$ for Amount. We can succinctly represent this as a set of atoms $\{\mathsf{Transfer}(t_4, w_4, w_3), \mathsf{Amount}(t_4, 15000)\}$.

Property Graphs. We define the property graph data model as follows. We assume pairwise disjoint countable sets of labels $\mathcal{L}$, keys $\mathcal{K}$, constants Const, node ids $\mathcal{N}$, and edge ids $\mathcal{E}$[1]. A *property graph* G is a tuple $(N, E, \rho, \lambda, \delta)$ with:

- $N \subseteq \mathcal{N}$ a finite subset of nodes; $E \subseteq \mathcal{E}$ a finite subset of edges;
- $\rho : E \to (N \times N)$ a total function that maps edges to pairs of nodes that represent the endpoints of edges;
- $\lambda : N \cup E \to 2^{\mathcal{L}}$ a total function that maps nodes and edges to finite (possibly empty) sets of labels;
- $\delta : (N \cup E) \times \mathcal{K} \to \mathrm{Const}$ a partial function that assigns constants to the keys in nodes and edges. For a given node or edge o, we call a key-value from $\{(k, v) \mid \delta(o, k) = v\}$ a *record*.

We say that for an edge $e \in E$ with $\rho(e) = (s, t)$, nodes s and t are *endpoints* of e, with s being the *source* and t the *target*. We sometimes also write $\mathrm{src}(e) = s$ and $\mathrm{tgt}(e) = t$.

To define provenance, we observe that property graphs can be captured by the relational model using the following schema:

$$\{\mathsf{Node}[nid], \mathsf{Edge}[eid, src, tgt], \mathsf{Label}_l[id] \text{ for every } l \in \mathcal{L},$$
$$\mathsf{KeyValue}_k[id, value] \text{ for every } k \in \mathcal{K}\}$$

Essentially, we have relation schema representing nodes and edges, as well as a relation for each possible label and property key. While this schema has an infinite number of relation schemas, for a given property graph, its relational schema is finite, since it has a finite number of labels and property keys. For a given property graph G, we write $\mathcal{S}(G)$ to denote its relational schema. This means that we can see any property graph as a (relational) instance of its schema.

Note that any relational instance that represents a property graph satisfies the following foreign key constraints:

- $\mathsf{Edge}[src], \mathsf{Edge}[tgt] \subseteq \mathsf{Node}[nid]$
- $\mathsf{KeyValue}_k[id] \subseteq \mathsf{Node}[nid] \cup \mathsf{Edge}[eid]$ for every $k \in \mathcal{K}$
- $\mathsf{Label}_l[id] \subseteq \mathsf{Node}[nid] \cup \mathsf{Edge}[eid]$ for every $l \in \mathcal{L}$

We call these the *relational-graph constraints*: edge endpoints must be nodes, and property keys and labels must refer to existing nodes and edges.

Example 3.2. The relational instance from Example 3.1 can be adjusted to represent a property graph. The Transfer represents an edge with a label, so it is is actually the Edge relation in combination with the $\mathsf{Label}_{\mathsf{Transfer}}$ relation. The Amount relation represents a property key, so it is the $\mathsf{KeyValue}_{\mathsf{Amount}}$ relation. Then, we need to complete the graph instance with nodes to make it satisfy the relational-graph constraints. Completing Example 3.1 gives us:

$$\{\mathsf{Node}(w_3), \mathsf{Node}(w_4), \mathsf{Edge}(t_4, w_4, w_3), \mathsf{Label}_{\mathsf{Transfer}}(t_4),$$
$$\mathsf{Label}_{\mathsf{Wallet}}(w_3), \mathsf{Label}_{\mathsf{Wallet}}(w_4), \mathsf{KeyValue}_{\mathsf{Amount}}(t_4, 15000)\}$$

The newly introduced nodes are labeled Wallet to make it represent part of G_{crypto} (Figure 1).

Subgraphs. A subgraph of a property graph can be understood primarily in two ways. First, we introduce a notion of subgraphs based on the sets of nodes and edges, we call such subgraphs *coarse*. Given a property graph $G = (N, E, \rho, \lambda, \delta)$, we define $G' = (N', E', \rho', \lambda', \delta')$ to be a (coarse) *subgraph* of G, denoted by $G' \subseteq_c G$, if G is a property graph and $N' \subseteq N$ and $E' \subseteq E$, where for every $e \in E' : \rho'(e) = \rho(e)$, for every $o \in N' \cup E' : \lambda'(o) = \lambda(o)$, and for every $k \in \mathcal{K}$: if $\delta(o, k)$ is defined, then $\delta'(o, k) = \delta(o, k)$. Second, in property graphs, subgraphs can not only be understood in terms of nodes and edges, but also with respect to their associated labels and properties. We define G' to be a (fine-grained) *subgraph* of G, denoted by $G' \subseteq_f G$, if

- $N' \subseteq N$; and $E' \subseteq E$ where for every $e \in E' : \mathrm{src}(e), \mathrm{tgt}(e) \in N'$;
- for every $e \in E' : \rho'(e) = \rho(e)$;
- for every $o \in N' \cup E' : \lambda'(o) \subseteq \lambda(o)$; and

[1] From a relational perspective, $\mathcal{L} \cup \mathcal{K}$ can be seen as a subset of the relation symbols $\mathbf{N}$ and $\mathcal{N} \cup \mathcal{E} \subseteq \mathbf{dom}$

- for every $o \in N' \cup E'$ and $k \in \mathcal{K}$: if $\delta'(o,k) = v$ is defined, then $\delta(o,k) = v$.

Throughout this paper, we clearly indicate which notion of subgraph must be understood. Note that if $G' \subseteq_c G$ then $G' \subseteq_f G$, but not vice versa. Furthermore, when looking at a property graph G as a set of atoms (a relational instance) of $\mathcal{S}(G)$, a subgraph G' is simply a subset of G such that G' satisfies the previously mentioned relational-graph constraints. It is easily verified that Example 3.2 is a subgraph of G_{crypto} (Figure 1).

3.2 Property Graph Queries

We proceed with the formal definition of property graph queries. Specifically, in order to explain provenance for GQL, we use a formalization called *Core-GQL* [14], which is like the underlying *algebra* of GQL operations.

Graph Pattern Matching. The foundation of graph query processing is *graph pattern matching* [10, 14, 22], which is based on *paths*. A path p is an alternating sequence of nodes and edges that starts and ends with a node. We explicitly denote a path of length k with $\text{path}(n_0, e_0, \ldots, e_{k-1}, n_k)$ where $n_0, \ldots, n_k \in \mathcal{N}$ and $e_0, \ldots, e_{k-1} \in \mathcal{E}$, where for each $0 \leq i < k$, $\text{src}(e_i) = n_i$ and $\text{tgt}(e_i) = n_{i+1}$. We write $nodes(p)$ to denote the set of node identifiers occurring in path p and $edges(p)$ to denote the edges. We next define *pattern matching expressions (PMEs)*. Assume an infinite set of variables $\mathcal{V}$. A PME is given by the following grammar:

$$\psi, \psi' \coloneqq (x) \mid \xrightarrow{x} \mid \xleftarrow{x} \mid \psi\psi' \mid \psi^{n..m} \mid \psi_{\langle\theta\rangle} \mid \psi + \psi'$$

with $x \in \mathcal{V}$ and θ a condition on the node or edge bound to that variable. The condition θ essentially allows for filtering out certain patterns based on the values of property keys of nodes and edges. It additionally encodes which labels are present in the nodes and edges. We adopt notation from Francis et al. [10] and sometimes write $(x : L)$ to denote that the node pattern with variable x requires matching nodes to have label L. In our syntax, this corresponds to a condition θ which requires that x has label L.

Importantly, the evaluation of a PME ψ in graph G, denoted by $[\![\psi]\!]_G$ is a set of *path-tuple pairs* (p, μ). We forgo the details of the semantics and refer to the literature for more details [10, 14] [2] . The inclusion of *paths* in the evaluation of PMEs reflects the prominence of paths within the context of property graph queries. Indeed, GQL directly supports querying for paths through so-called *path variables* [21].

Example 3.3. The `MATCH-WHERE` clause of query Q_{transf} from the introduction can be written as the PME:

$$(p : \text{Person}) \xrightarrow{o:\text{Owns}} (w_1 : \text{Wallet}) \xrightarrow{t:\text{Transfer}} (w_2)_{t.amount>4000}$$

Evaluating this PME on the graph G_{crypto} (Figure 1) results in the paths and relation (set of tuples):

path	p	w_1	w_2	t	o
$(n_1 e_3 n_7 e_8 n_6)$	n_1	n_7	n_6	e_8	e_3
$(n_2 e_2 n_5 e_6 n_4)$	n_2	n_5	n_4	e_6	e_2

[2] Following [14], we omit group/path variables in PMEs.

Core-GQL. Importantly, in property graph query languages, pattern matching is restricted to match only *finite* paths. This restriction is made explicit by *path modes* [11]. We consider the path modes "acyclic" and "trail." The "acyclic" path mode only matches paths with no repeated nodes, while the "trail" path mode only matches paths with no repeated edges. To consider this feature, we define a *graph pattern matching query (PMQ)* Q as an expression with grammar: $Q, Q' \coloneqq M\psi \mid Q \bowtie Q'$ where M is one of the path modes. PMQs are a subset of the GPC [10]. Throughout this paper, if no explicit path mode is given for a PMQ, we assume it is "trail."

The evaluation of a PMQ Q in a graph G, denoted by $[\![Q]\!]_G$, results in a relation containing node- and edge-ids. Specifically, it is a relation with schema $R_Q[\text{FV}(Q)]$, i.e., the variables used in Q are the attributes of the relation, where each result tuple binds the variables to nodes or edges.

In reality, GQL queries can also return constant values as results—not only nodes and edges. For that purpose, we define a *PMQ with Output (PMQO)* to be a pair (Q, Ω) with Q being a PMQ, and Ω being a set of *variable literals*; either a variable from $\mathcal{V}$ or an expression of the form $x.k$ with $x \in \mathcal{V}$ and $k \in \mathcal{K}$. The set Ω defines the output attributes of the relation resulting from evaluating a PMQO. Formally, the evaluation of a PMQO (Q, Ω) in a graph G, denoted by $[\![(Q, \Omega)]\!]_G$, results in a relation with schema $R_{(Q,\Omega)}[\Omega]$.

Example 3.4. Consider again query Q_{transf} from the introduction. When disregarding the renaming (i.e., the "AS" keyword), we can express it as a PMQO. Let Q be the PME from Example 3.3. We can then write query Q_{transf} as $(Q, \{p.name, p.riskScore\})$. The output variable literals $p.name$ and $p.riskScore$ specify the exact records that we are interested in with regards to the nodes and edges matched with the PME Q.

Finally, as shown by Gheerbrant et al. [14], the foundation of property graph queries lies the relational algebra. More specifically, it is the relational algebra over database schemas consisting of relation schemas that result from PMQOs.

Example 3.5. We can express query Q_{transf} from the Introduction in full with the help of the standard relational algebra renaming operator: $\rho_{p.name \mapsto name; p.riskScore \mapsto riskScore}(Q')$ where Q' is the PMQO from Example 3.4.

While Example 3.5 only needs the renaming operator ρ, generally, we need the full power of relational algebra to express GQL [14]. As part of our contribution, we provide an algorithm that shows how to generally translate a GQL query into Core-GQL in Section 5.

4 WHY-PROVENANCE FOR GRAPH QUERIES

We define why-provenance for property graph queries based on the underlying algebra Core-GQL. The goal of why-provenance is to identify subgraphs that contribute to a query answer. In Section 3 we presented GQL as built from two major components: pattern matching and relational algebra. We define why-provenance for pattern matching expressions (PMEs) and show its propagation across GQL operators, ensuring that provenance is systematically preserved throughout query execution. In particular, we focus on defining why-provenance for the *monotone* fragment of Core-GQL. Why-provenance for non-monotone operators is out of scope for this work.

Classically, why-provenance is defined in terms of minimal subinstances of the database. Such instances are called *sufficient* subinstances since they represent part of the database that is enough to reproduce a query answer [15]. Similarly, we can define sufficient subgraphs (also called *witnesses* in the why-provenance literature).

Definition 4.1. Given a query Q, a graph G, and an answer tuple $\mu \in [\![Q]\!]_G$, we say a subgraph $G' \subseteq G$ is *sufficient* if $\mu \in [\![Q]\!]_{G'}$.

The intuition behind sufficiency breaks down in presence of non-monotone queries; a sufficient subgraph may no longer produce a query result after adding data. Ultimately, why-provenance for a query answer can be characterized as the set of minimal sufficient subgraphs, also known as the set of *minimal witnesses*.

Definition 4.2. We define the set of minimal witnesses as follows:

$$\mathsf{MWIT}_{\subseteq}(Q, G, \mu) := \{G' \subseteq G \mid \mu \in [\![Q]\!]_{G'} \text{ and } \nexists G'' \subseteq G' \text{ s.t. } \mu \in [\![Q]\!]_{G''}\}$$

where $\subseteq$ is either $\subseteq_c$ (coarse) or $\subseteq_f$ (fine-grained).

Definition 4.2 is parameterized by the kind of subgraph we consider. Indeed, depending on whether we consider *coarse* (subsets of nodes and edges) or *fine-grained* (cf. Section 3) subgraphs, the set of minimal witnesses is different.

Example 4.3. Consider the query Q_{rich} asking for the people with wallets where the balance is greater than 6000. Formulated in GQL:

```
Q_rich : MATCH (x1:Person)-[o:Owns]->(x2:Wallet)
         WHERE x2.balance > 6000 RETURN x1.name AS name
```

Executing this query on the running example (Figure 1) results in one answer tuple $\mu := \{name \mapsto Alice\}$. The (coarse) $\subseteq_c$-minimal sufficient subgraph is G_{alice} (Figure 1). It simply contains the nodes n_1 and n_7 together with their complete records as well as edge e_3. However, the (fine-grained) $\subseteq_f$-minimal sufficient subgraph does not include the complete records. It only needs to include the name of the person and the balance of their wallet (k_1, k_{17}), as well as the relevant labels (l_1, l_7, l_{11}).

Why-provenance is typically discussed in relation to minimal witnesses. While the idea of minimal witnesses (Definition 4.2) is straightforward, it is not immediately clear how to effectively compute it. Why-provenance approaches minimal witnesses by using a *syntactic* characterization [6, 13]; the definition relies on the syntax of the query language instead of the semantics. For graph queries, however, it is instead natural to consider *paths* for characterizing provenance. Such characterization is also done in works related to provenance for regular path querying and graph algorithms [17, 26]. In case of GQL, we directly use the paths given by the pattern matching expressions (PMEs) to define provenance.

Recall that graph queries consist of multiple components: pattern matching expressions (PMEs), pattern matching queries (PMQs), pattern matching queries with output (PMQOs) and, finally, Core-GQL, which applies relational algebra on PMQOs.

Pattern Matching Expressions. First for PMEs, the most fundamental building block of Core-GQL, the evaluation directly gives us the path.

Definition 4.4. Given a PME ψ, graph $G := (N, E, \rho, \lambda, \delta)$, and a result $(\mu, p) \in [\![\psi]\!]_G$, we define $\mathsf{WHY}_{\subseteq_c}(\psi, G, (\mu, p))$ to be the subgraph $(nodes(p), edges(p), \rho, \lambda, \delta)$.

Example 4.5. The pattern in query Q_{Qrich} from Example 4.3 can be expressed by the PME ψ_{rich}:

$$(x_1 : \text{Person}) \xrightarrow{o:\text{Owns}} (x_2 : \text{Wallet})_{x_2.balance>6000}$$

which matches the pattern with tuple $\mu := \{x_1 \mapsto n_1; o \mapsto e_3; x_2 \mapsto n_7\}$ as well as the path $p := n_1 e_3 n_7$. According to Definition 4.4, this results in the graph with the nodes $nodes(p) := \{n_1, n_7\}$ and edges $edges(p) := \{e_3\}$, together with the complete records. This is exactly the (coarse) $\subseteq_c$-minimal sufficient subgraph G_{alice}, as described in Example 4.3.

Since the path directly represents all the necessary nodes and edges to recreate it, we also know it represents a sufficient subgraph. Furthermore, it is minimal: removing one of the nodes or edges mentioned in the path no longer allow us to recreate that path. This gives us exactly the coarse minimal witness.

While coarse subgraphs provide a structural foundation for why-provenance, they often include information that is not strictly necessary. By operating at the level of entire nodes and edges, coarse subgraphs do not account for specific subsets of records within nodes or edges. To achieve a more refined provenance, we introduce *fine-grained* why-provenance. The goal is to include only the relevant components of the nodes and edges matched by the path, rather than their entire state.

Example 4.6. Continuing with the PME ψ and its result (μ, p) from Example 4.5, the why-provenance results in the subgraph G_{alice} which contains the node n_1 completely. However, to reproduce the pattern matching result, we do not need to include the complete node. For example, the record that states that the city of n_1 is "Paris" is irrelevant for pattern ψ. In fact, the only information about n_1 we need to reproduce the result is that n_1 has the label Person.

Capturing only relevant node and edge labels and properties for a query result cannot be achieved using matched paths alone. Therefore, to define fine-grained why-provenance, we specify, for each variable used in the PME, the expected property keys and labels for values bound to those variables. The intuition is that the query indicates the expected "shape" of a node or edge: for any variable in the query, the property keys and labels used with that variable shows the relevant shape. We call this "shape" for variable x in PME ψ the *variable schema* of x in ψ, denoted by $v_\psi(x)$. We use the variable schema to define fine-grained why-provenance.

Definition 4.7. Given a PME ψ, graph G, and a result $(\mu, p) \in [\![\psi]\!]_G$, we define $\mathsf{WHY}_{\subseteq_f}(\psi, G, (\mu, p))$ as the set

$$\{\mathsf{Node}(n), \mathsf{Edge}(e), \mathsf{Node}(n') \mid nen' \text{ is a sequence in } p\}$$
$$\cup \{\mathsf{Label}_l(o) \mid \mu(x) = o \wedge l \in v_\psi(o) \wedge l \in \lambda(o)\}$$
$$\cup \{\mathsf{KeyValue}_k(o, v) \mid \mu(x) = o \wedge k \in v_\psi(o) \wedge \delta(o, k) = v\}$$

This definition allows us to have smaller subgraphs than before.

Example 4.8. The PME ψ from Example 4.5 has variable schema $v_\psi(x_1) := \{\text{Person}\}$, $v_\psi(o) = \{\text{Owns}\}$ and $v_\psi(x_2) :=$

{Wallet, *balance*}. These variable schemas tell us which records and labels must be included in the fine-grained why-provenance. Indeed, now the minimal witness is

$$\{\mathsf{Node}(n_1), \mathsf{Node}(n_7), \mathsf{Edge}(e_3), \mathsf{Label}_{\mathsf{Person}}(n_1), \mathsf{Label}_{\mathsf{Owns}}(e_3), \mathsf{Label}_{\mathsf{Wallet}}, \mathsf{KeyValue}_{balance}(n_7, 7000)\}$$

Pattern Matching Queries. The end-goal is to get provenance for Core-GQL, not only for PMEs. The next step towards this goal is to define provenance for pattern matching queries (PMQs). Essentially, PMQs add three operations to PMEs. First, PMQs project away the paths in the result; the evaluation of a PMQ is an answer tuple μ. Second, PMQs introduce path modes M, i.e. *acyclic* or *trail* (queries of the form $M\psi$), effectively filtering which types of paths are allowed. Lastly, PMQs add the pattern join construct (i.e., queries of the form $Q \bowtie Q'$). We define why-provenance as:

Definition 4.9. Given PMQs Q, graph G, and a result $\mu \in [\![Q]\!]_G$, if Q is of the form $M\psi$, we define $\mathsf{WHY}_{\subseteq}(Q, G, \mu)$ as the set of subgraphs

$$\{\mathsf{WHY}_{\subseteq}(\psi, G, (p, \mu)) \mid \exists p : p \text{ is } M;\ (p, \mu) \in [\![\psi]\!]_G \text{ and } \quad (i)$$
$$\nexists p' : p' \text{ is } M;\ (p', \mu) \in [\![\psi]\!]_G \text{ and } \quad (ii)$$
$$edges(p') \subseteq edges(p)\}$$

and if Q is of the form $Q' \bowtie Q''$ we define $\mathsf{WHY}_{\subseteq}(Q, G, \mu)$ as $\mathsf{WHY}_{\subseteq}(Q', G, \mu') \uplus \mathsf{WHY}_{\subseteq}(Q'', G, \mu'')$, where tuple μ' (resp. μ'') is μ restricted to the variables of Q' (resp. Q'').

Unlike Definition 4.7, there are now multiple minimal witnesses: the why-provenance provides a set of subgraphs. Specifically, nearly every path possibly gives us a witness. Indeed, there are two conditions on paths in order for them to be considered for constructing a witness. First, (i) it must be a path for the associated PME. Second, (ii) we only want paths that are subset-minimal with relation to the edges. This second requirement ensures $\subseteq_c$-minimal witnesses, but is a computationally expensive operation. We show the intuition behind the requirement (ii) with the following example.

Example 4.10. Consider again graph G_{crypto} (Figure 1), and a query that identifies all pairs of wallets connected through at most four transfers (undirected, i.e., either incoming our outgoing):

```
MATCH (x:Wallet) (()-[:Transfer]-()){1,4} (y:Wallet) RETURN x,y
```

One of the results is $(x \mapsto n_6, y \mapsto n_7)$. The why-provenance of this answer is based on paths. The `MATCH` pattern from the query matches the following paths (among others): $p_1 = n_6 e_8 n_7$, $p_2 = n_6 e_7 n_5 e_9 n_7$, and $p_3 = n_6 e_7 n_5 e_6 n_4 e_5 n_5 e_9 n_7$. The path p_1 is edge-subset minimal and is therefore considered when computing provenance. So is path p_2, but not p_3 since $edges(p_2) \subseteq edges(p_3)$ (violating the edge-subset minimal requirement). Note that there is a difference between *shortest paths* (i.e., of minimal length) and *edge-subset minimal* paths. With shortest paths we would miss p_2.

Pattern Matching Queries with Output. Core-GQL is defined as relational algebra over pattern matching queries with output (PMQOs). Therefore, if we have provenance for PMQOs, we can leverage why-provenance for relational algebra to obtain provenance for Core-GQL. PMQOs are PMQs where we additionally specify for each variable what values for property keys need to be outputed (i.e., the output Ω c.f. Section 3). Intuitively, this output set Ω essentially adds to the variable schema. Now, we do not only consider the property keys of PMEs as part of the variable schema, but also the property keys from the output Ω. Thus, provenance for PMQOs is simply Definition 4.9 where the variable schemas used in Definition 4.7 are extended to include the property keys from Ω.

Example 4.11. Consider again query Q_{rich} (Example 4.3). We can express this query as a PME given in Example 4.5 with path mode *trail* (the default mode in GQL), and output $\Omega = \{x_1.name\}$. If we want to compute the fine-grained why-provenance, we first deduce the variable schema for the variables x_1 and x_2. In addition to the variable schemas for these variables given in Example 4.8 the variable schema for x_1 also includes the *name* property key since this is necessary for the output of the query: it is mentioned as an *output* variable.

Why-Provenance for Core-GQL. For Core-GQL, we can now adopt the definition of why-provenance from literature, with the specific feature that instead of operating on relations, the provenance is computed based on the relations that are dynamically given by PMQOs. Analogously, the why-provenance for Core-GQL is dynamically build up from why-provenance for PMQOs. We illustrate this with the following example.

Example 4.12. Consider query Q_{transf} from the introduction, and query Q_{rich} (Example 4.3). We can combine these two queries into a more complex GQL query:

Q_{transf} `NEXT` Q_{rich} `NEXT RETURN name, riskScore`

This can be written in Core-GQL as the join of two PMQOs:

$$\pi_{name,riskScore}(\rho_{p.name \mapsto name; p.riskScore \mapsto riskScore}(R_{\psi_1,\Omega_1}) \bowtie \rho_{x.name \mapsto name}(R_{\psi_{rich},\Omega_2}))$$

where $\Omega_1 := \{p.name; p.riskScore\}$, $\Omega_2 := \{x.name\}$, ψ_1 is

$$(p : \mathsf{Person}) \xrightarrow{:\mathsf{Owns}} (w_1 : \mathsf{Wallet}) \xrightarrow{t:\mathsf{Transfer}} (w_2)_{t.amount>4000}$$

and ψ_{rich} is the pattern from Example 4.5. The why-provenance for this composite result is computed by substituting relational tuples with the why-provenance of their contributing PMQOs. In this case, Alice and Bob are rich (i.e., appear in the result of $[\![Q_{rich}]\!]_G$). Their result tuples are annotated with subgraphs G_{alice} and G_{bob} respectively (Figure 1) under *coarse* provenance. Under fine-grained provenance, these are further reduced to the smaller subsets of labels and properties required to reproduce the result.

5 EFFICIENT COMPUTATION OF PROVENANCE FOR GQL

In this section, we introduce the first query-rewriting-based approach for computing why-provenance in GQL. In Section 4 we defined provenance for Core-GQL. Now, we show how to practically compute this provenance. Our approach supports provenance for queries with pattern matching expressions (PMEs) and supports `MATCH`, `FILTER`, and `RETURN` clauses, `CALL`, and `NEXT` statements, as well as the set operator `UNION`. Furthermore, since our approach is based on rewriting, it can be used on any GQL-compliant graph database system.

We compute why-provenance using annotations, i.e., metadata assigned to every node, edge, property, and label [18]. While

Algorithm 1 Algorithm for why-provenance computation

Require: A property graph G, a GQL query Q, and an annotation function $\mathcal{A}$ for G

1: $T_\emptyset \leftarrow \{()\}$ // Initialize unit relation

// **Phase 1: Translate GQL query to Core-GQL tree**

2: $T_Q, \mathcal{V}_{out} \leftarrow$ constructCoreGQLTree$(Q, T_\emptyset)$

// **Phase 2: Rewrite GQL query**

3: adornCoreGQLTree(T_Q)

4: $Q^* \leftarrow$ rewriteQuery$(Q, T_Q, \mathcal{A})$

// **Phase 3: Decode Provenance Encodings**

5: *results* $\leftarrow$ decodeProvenance$(Q^*, G, \mathcal{V}_{out})$

6: **return** *results*

our approach is agnostic to the specific scheme (e.g., generic identifiers as in Figure 1, source information, or security levels), we define a general annotation function $\mathcal{A}(x)$ for concreteness. This function maps each element to a unique identifier:

$$\mathcal{A}(x) = \begin{cases} ID(x), & \text{if } x \in N \cup E \\ ID(o).k, & \text{if } x = o.k;\ o \in N \cup E,\ (o,k) \in dom(\delta) \\ ID(o) : l, & \text{if } x = o : l;\ o \in N \cup E,\ l \in \lambda(o) \\ \varnothing & \text{otherwise} \end{cases}$$

Given a property graph G, a GQL query Q, and an annotation function $\mathcal{A}$ for G, our approach annotates each result μ of the query evaluation $[\![Q]\!]_G$ with fine-grained why-provenance, which is represented as a set of annotation sets $\mathcal{A}(\mathsf{WHY}_{\subseteq_f}(Q, G, \mu))$, where each set represents a subgraph of G that contributed to the generation of μ (Example 4.12). We primarily focus on the computation of *fine-grained* why-provenance, $\mathsf{WHY}_{\subseteq_f}$, which we will refer to simply as why-provenance, WHY, in the remainder of this section.

To compute why-provenance, our approach rewrites a query Q into Q^*. The rewritten query Q^* encodes the why-provenance of a query result into output variables over (possibly multiple) result tuples, using the variables present in the original query Q along with their associated property keys and labels. We refer to the resulting representation of provenance as the *provenance encoding*. Intuitively, decoding the provenance encoding of Q^* yields the why-provenance for the query results of the original query Q.

Given a property graph G, a GQL query Q, and an annotation function $\mathcal{A}$ for G, we compute why-provenance by proceeding through three major phases, which are outlined in Algorithm 1:

(1) The first step is to parse query Q into a (syntax tree of) Core-GQL T_Q (Section 5.1) so that this representation can guide the rewriting.
(2) Rewrite Q into a provenance-enhanced query Q^* that additionally computes provenance information (Section 5.2).
(3) Execute Q^* on G and decode the why-provenance for each query result (Section 5.3)

We discuss each step in more detail in the following subsections.

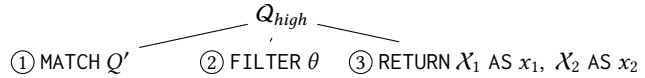
5.1 GQL to Core-GQL

This section details Phase 1 of Algorithm 1: the translation of a GQL query into a Core-GQL (syntax) tree. With our approach, we encode why-provenance within output variables called *provenance variables*. We compute these provenance variables and their propagation through subqueries using the Core-GQL tree. Throughout all of Algorithm 1, we assume that the input GQL query Q is represented as a syntax tree given by the syntax described in the GQL ISO standard [21]. Let us first present an example to familiarize the reader with GQL queries represented as a syntax tree.

Example 5.1. Consider the following GQL query Q_{high}, which identifies people or companies with high risk scores whose accounts transferred funds to accounts that already had larger balances:

```
Q_high : MATCH (a:Person|Company)-[]->(s:Wallet)-[:Transfer]->(r)
         FILTER a.riskScore >=0.7 AND s.balance < r.balance
         RETURN a.name AS name, a.city AS city
```

The parsed syntax tree of Q_{high} using the GQL grammar is shown below. For brevity, we present only its clause-level structure.



where

$$Q' \coloneqq (a : Person|Company) \rightarrow (s : Wallet) \xrightarrow{:Transfer} (r),$$
$$\theta \coloneqq \langle a.riskScore >= 0.7 \wedge s.balance < r.balance \rangle,$$
$$\mathcal{X}_1 \coloneqq a.name,\ x_1 \coloneqq name,\ \mathcal{X}_2 \coloneqq a.city, \text{ and } x_2 \coloneqq city$$

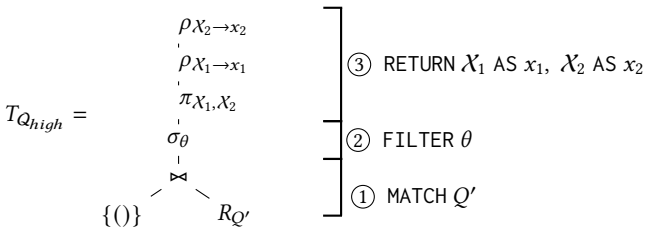
We now describe the process of constructing such a Core-GQL tree from a GQL query (the constructCoreGQLTree function in Algorithm 1). This process is a bottom-up construction that follows the logical structure of the input query:

- The construction of the Core-GQL tree T_Q starts with the unit relation ($T_\emptyset$), a single empty tuple representing the base state for GQL query evaluation.
- When a MATCH clause is encountered, the corresponding pattern matching query (Q) is mapped to a relational schema node (R_Q) and joined with the current root of T_Q. Join pattern matching queries ($Q' \bowtie Q''$) are recursively transformed and joined.
- The FILTER clause with condition θ is translated into a selection node (σ_θ) over the current root.
- The RETURN clause with output attributes U and renamings X is transformed into a projection (π_U) and renaming (ρ_X) operation. During this step, output attributes are collected into the set $\mathcal{V}_{out}$. Crucially, due to the nature of GQL queries, subqueries always end with a RETURN clause.
- The NEXT operator and CALL statements induce a recursive construction of subqueries. The set operator UNION is handled similarly through recursive transformation of its subqueries.

Crucially, each node in the resulting Core-GQL tree preserves the mapping with the input query. These mappings between the input query and the resulting Core-GQL tree are necessary to maintain a linkage for the subsequent provenance rewriting phase. Concretely, we maintain a persistent mapping between each clause in Q (e.g. RETURN) and its corresponding algebra node in T_Q (e.g. a projection). This linkage is later used in Algorithm 3 to correctly perform the rewriting.

Example 5.2. We continue with Example 5.1 to illustrate the mapping from clauses to algebra. Starting from the unit relation $T_\emptyset$, we construct the algebra tree $T_{Q_{high}}$ by transforming each GQL clause into its algebraic equivalent. The numbers at each node of Q_{high}

indicate the traversal order. At node ①, we introduce a join ($\bowtie$) between the base $T_\emptyset$ and the relational schema $R_{Q'}$ representing the pattern match. At node ②, we apply a selection node (σ_θ) to the current root to enforce the query conditions. At node ③ the procedure finalizes the tree with projection and renaming nodes, while simultaneously defining the output variables $\mathcal{V}_{out} = \{\text{name}, \text{city}\}$. The resulting Core-GQL tree is shown below.



The leaves of the Core-GQL tree contain either the unit relation {()} or the relational schemas R_Q of the PMQs. In our example, $T_{Q_{high}}$ resolves to {()} and $R_{Q'}$. The creation of the Core-GQL tree is crucial: it provides a structured representation of the query that guides the rewriting process to propagate provenance accurately.

Algorithm 2 AdornCoreGQLTree

Require: A Core-GQL tree node M
Ensure: Propagate and store provenance variables

procedure AdornCoreGQLTree(M)
  ReplaceAnonVars(M)
  PropagateQueryVars(M)
  AlignUnions($M, \emptyset$)
procedure ReplaceAnonVars(M)
  for all leafs R_Q of M **do**
    Replace anon. vars. in Q with fresh vars. $fresh(f)$.
procedure PropagateQueryVars(M)
  $v \leftarrow \emptyset$ // Initialize query schema
  for all children M' of M **do**
    $v[new] \leftarrow v \cup$ PropagateQueryVar(M')
  $v[new] \leftarrow v[new] \cup \{fresh(t) \mapsto v \mid v$ is a var.sch.el. of $M\}$
  $v[pass] \leftarrow v[pass] \cup \{f \mapsto f \mid f$ is a fresh var. from $M\}$
  if M is a projection π **then**
    $M.new \leftarrow v[new]$; $M.pass \leftarrow v[pass]$
    $v[pass] \leftarrow v[pass] \cup v[new]$
    $v[new] \leftarrow \emptyset$
  return v
procedure AlignUnions(M, v)
  if $v = \emptyset$ **then**
    for all children M' of M **do**
      $v \leftarrow v \cup$ AlignUnions($M', \emptyset$)
    if M is a projection π_X **then return** X
    if M is a union $M' \cup M''$ **then**
      AlignUnions(M', v); AlignUnions(M", v)
      return v
  else if M is a projection **then**
    $M.none \leftarrow v - (M.new \cup M.pass)$
  else
    for all children M' of M **do**
      AlignUnions(M', v)

5.2 Query Rewriting

In this section, we present our approach for query rewriting (Algorithm 1, Phase 2). Given a query Q, we rewrite it into a query Q^* that explicitly encodes provenance in its output variables. This allows the database engine to carry provenance annotations alongside the final results of Q in Q^*. Before we present the rewriting algorithm itself, we first present the way that provenance is encoded in Q^*.

5.2.1 Provenance Encoding for GQL Queries. GQL queries allow variables to range over nodes, edges, constant values, as well as collections of them, such as lists and paths. Importantly, GQL features path variables which allow easy access to paths matched by patterns. We leverage these richer variable types for provenance encoding. We use **path variables** to encode provenance of nodes and edges as lists, as well as **atomic** and **group variables** to encode the provenance of properties and labels of the bound graph elements as atomic values and lists, respectively.

Example 5.3. Recall the match clause of Q_{high} in Example 5.2.

```
MATCH p0 = (a:Person|Company)-[]->(s:Wallet)-[:Transfer]->(r)
```

By assigning this match clause path variable p_0 (marked in orange), all matched nodes and edges can be accessed uniformly through the paths bound to p_0, including the edges matched to edge pattern `-[]->`, which has no variable (i.e., an anonymous variable).

The following example illustrates the provenance encoding created by our query rewriting and its rationale.

Example 5.4. Recall query Q_{high} from Example 5.2. Its evaluation over G_{crypto} (Figure 1), $[\![Q_{high}]\!]_{G_{crypto}}$, and the why-provenance annotations $\text{WHY}(Q_{high}, G_{crypto}, \mu)$ for each tuple is given below.

	name	city	$\text{WHY}(Q_{high}, G_{crypto}, \mu)$
μ'	"Alice"	"Paris"	$\{\{n_1, n_4, n_5, e_1, e_5, l_1, l_4, l_{12}, k_1, k_2, k_3, k_{11}, k_{13}\}, \{n_1, n_6, n_7, e_3, e_8, l_1, l_7, l_{15}, k_1, k_2, k_3, k_{15}, k_{17}\}\}$
μ''	"HSBC"	"London"	$\{\{n_3, n_6, n_7, e_4, e_7, l_3, l_6, l_{14}, k_7, k_8, k_9, k_{13}, k_{15}\}\}$

We illustrate the provenance encoding of $\text{WHY}(Q_{high}, G_{crypto}, \mu)$ in Table 1. Apart from the output variables defined in the original query (**name** and **city**), there are a number of other–newly introduced–variables which encode the provenance. We call these variables *provenance variables* which are indicated in orange. The variables $l_P(a)$, $l_C(a)$, $l_W(a)$, and $l_T(a)$ represent the node and edge labels *Person*, *Company*, *Wallet*, and *Transfer*, respectively. The other provenance variables represent the paths and specific property keys needed for the provenance encoding (e.g., *a.riskScore*).

Some of the provenance variables are based on existing variables from the input query, others are freshly introduced. In Q_{high}, we introduce a path variable p_0 for the `MATCH` clause from Example 5.2 and a variable t for the edge pattern $\xrightarrow{:Transfer}$.

Then, for each tuple $\mu \in [\![Q_{high}]\!]_{G_{crypto}}$, we encode the provenance of nodes and edges (n_i and e_j) in $\text{WHY}(Q_{high}, G_{crypto}, \mu)$ using the path variable p_0, and the provenance of associated labels and properties (l_p and k_q) using the variables a, s, r, and t. Note that the provenance annotation corresponding to the edge pattern $\to$ in Q' is captured by p_0 even without a variable, e.g. e_1 is encoded within row 1, variable p_0 in Table 1. We detail the decoding of the provenance variables in Section 5.3.

	name	city	p_0	$l_P(a)$	$l_C(a)$	$l_W(a)$	$l_T(t)$	$a.riskScore$	$s.balance$	$r.balance$	$a.name$	$a.city$
μ_1^*:	"Alice"	"Vienna"	$[n_1, e_1, n_4, e_5, n_5]$	l_1		l_4	l_{12}	k_3	k_{11}	k_{13}	k_1	k_2
μ_2^*:	"Alice"	"Vienna"	$[n_1, e_3, n_7, e_8, n_6]$	l_1		l_7	l_{15}	k_3	k_{17}	k_{15}	k_1	k_2
μ_3^*:	"HSBC"	"London"	$[n_3, e_4, n_6, e_7, n_5]$		l_3	l_6	l_{14}	k_9	k_{15}	k_{13}	k_7	k_8

Table 1: Why-provenance of the Q_{high} result (Example 5.4), encoded according to the proposed provenance encoding principles.

Algorithm 3 REWRITEQUERY

Require: A GQL syntax tree N; Core-GQL tree T; annotation function $\mathcal{A}$
Ensure: Provenance rewritten query Q^*

1: **for all** nodes N' in N **do**
2: Let M be the node in T corresponding to N'.
3: **if** N' is a MATCH clause **then**
4: Add fresh variables f from M to N'.
5: **else if** N' is a RETURN clause **then**
6: **for all** variable mappings $t \mapsto v$ in $M.pass$ **do**
7: Add t as a return variable in N'
8: **for all** variable mappings $t \mapsto v$ in $M.new$ **do**
9: Add "$P_{GQL}(\mathcal{A}, v)$ AS t" to N'
10: **for all** variable p in $M.none$ **do**
11: Add "[] AS p" to N'

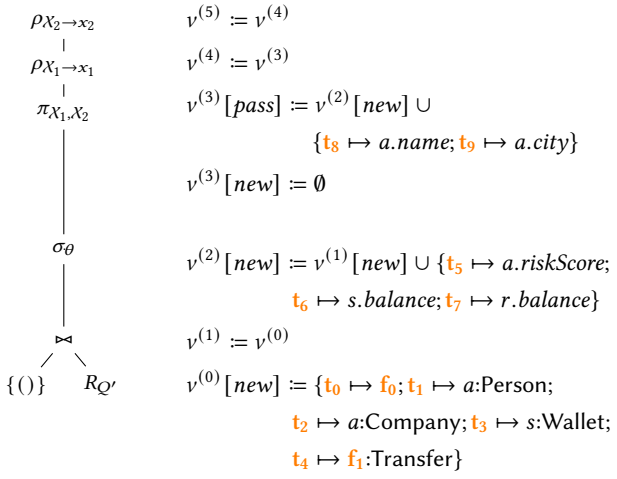
5.2.2 Query Rewriting Procedure. The query rewriting phase itself consists of two parts: first, the computation of variable propagation over the Core-GQL tree with ADORNCOREGQLTREE (Algorithm 1, line 3), followed by executing the rewriting of the GQL query itself with REWRITEQUERY (Algorithm 1, line 4).

ADORNCOREGQLTREE (Algorithm 2). The variable propagation is represented by adorning the Core-GQL tree with variable schema information (cf. Section 4). The procedure is split over three subprocedures.

First, **REPLACEANONVARS, (lines 5–7)** introduces fresh variables into the pattern matching expressions in place of anonymous and absent path variables. Effectively, everywhere a variable *can* appear, one will be introduced by this procedure. As a convention, in this paper, we use fresh variables of the form f_i (indicated by $fresh(f)$, line 7) for this purpose. This step is essential to enable the variable propagation of anonymous variables in the next step.

Second, **PROPAGATEQUERYVARS (lines 8–17)** propagates provenance variables through the query. The propagation is supported by a data structure representing the variable schema for the complete query. We call this data structure the *query schema* v. It consists of two components: $v[new]$ and $v[pass]$. Both components are a set of variable bindings of the form $a \mapsto b$ where a is a (node or edge) variable and b is a *variable schema element* (e.g., the variable schema element $a : Person$ represents that variable a has label $Person$). Component $v[new]$ is used in REWRITEQUERY (Algorithm 3) to construct the correct fine-grained provenance annotations. Component $v[pass]$ is also a set of bindings meant for propagation (to "pass" them on to the outer query). This procedure traverses the Core-GQL tree in post-order. When it encounters a Core-GQL tree node M that contains variable schema elements, it creates a fresh variable name for it. As a convention, freshly introduced variables are of the form t_i (indicated by $fresh(t)$, line 12). When the node M is a projection, we store the query schema information (line 15).

Example 5.5. We revisit query Q_{high} from Example 5.2 to demonstrate the adornment procedure (Algorithm 2, lines 5–17). We illustrate the process using the figure below.



where

$$Q' := f_0 = (a : Person|Company\}) \to (s : Wallet) \xrightarrow{f_1:Transfer} (r),$$
$$\theta := \langle a.riskScore >= 0.7 \wedge s.balance < r.balance \rangle,$$
$$X_1 := a.name,\ x_1 := name,\ X_2 := a.city, \text{ and } x_2 := city$$

The REPLACEANONVARS procedure introduces the fresh variables f_0 for the path, and f_1 for the anonymous edge. Then, the PROPAGATEQUERYVARS procedure traverses the Core-GQL tree bottom-up, and adds the new fresh variables $t_0, \ldots, t_4$ to the initial query schema $v^{(0)}$ for the various schema elements present in the pattern matching expression ($R_{Q'}$). Similarly, when encountering the selection (σ_θ) we add fresh variables $t_5, \ldots, t_7$ to $v^{(2)}$, etc. Importantly, when encountering the projection π_{X_1,X_2}, we store the query schema (Algorithm 2, line 15) and transfer the "new" variable bindings to the "pass" component of the query schema. The remaining renamings have no effect on the query schema. Indeed, renamings operate on the original query variables, and for the purposes of provenance propagation we always introduce fresh variables.

We continue with the description of the third and last subprocedure of ADORNCOREGQLTREE. GQL–like other query languages–has a syntactic restriction on the usage of the union-operator: its subqueries must have the same output variable signature. We call this restriction *union-compatibility*. The introduction of fresh variables in the previous two procedures can end up violating union-compatibility. Indeed, the introduction of fresh variables in the operands of the union operator must be handled with care. In the simplest case, if Q_1 and Q_2 are two match-subqueries of a union operator, we add the freshly introduced provenance variables of Q_1

to Q_2 and vice-versa. However, when Q_1 and Q_2 are more complex, alignment must be carefully applied. The procedure **alignUnions (lines 18–29)** traverses the tree in post-order and aligns the unions recursively. Importantly, we store the aligned variables in the projection nodes (line 26). These variables will later be used by the rewriting procedure to make sure they are introduced correctly.

rewriteQuery (Algorithm 3). Finally, we are ready to execute the rewriting on the original input query. This is handled by rewriteQuery (Algorithm 3). This procedure essentially goes through the input query and applies the necessary changes based on the adorned Core-GQL tree. There are two places in the input query that will be altered. First, the fresh variables introduced by replaceAnonVars will be added to the match-clauses. Second, the appropriate variables will be added to the return-clauses. This is based on the information stored in the corresponding projection nodes from the adorned Core-GQL tree. Recall that in the adornment procedure three kinds of variables are stored: *new*, *pass*, and *none* variables. Each are treated separately (lines 6-11). First, *pass* variables need to simply be passed on, so they are simply added as output variables in the return-clause. Second, *none* variables were introduced for union alignment. The necessary variables are bound elsewhere in the query, so if they appear as "none", they are added as empty variables (i.e., the empty list is bound). Finally, the *new* variables are important for the provenance encoding. Depending on the precise *kind* of variable, certain *rewriting rules* are applied determined by $P_{GQL}(\mathcal{A}, v)$ function where $\mathcal{A}$ is the annotation function and v is a variable schema element. It is defined as follows: $P_{GQL}(\mathcal{A}, v) =$

$$\begin{cases} [\mathcal{A}(a) \texttt{ FOR } a \texttt{ IN } v] & \text{if } v \text{ is a path variable} \\ \mathcal{A}(x.k) & \text{if } v = x.k \text{ and } x \text{ is an atomic variable} \\ [\mathcal{A}(a.k) \texttt{ FOR } a \texttt{ IN } x] & \text{if } v = x.k \text{ and } x \text{ is a group variable} \\ \mathcal{A}(x : l) & \text{if } v = x : l \text{ and } x \text{ is an atomic variable} \\ [\mathcal{A}(a : l) \texttt{ FOR } a \texttt{ IN } x] & \text{if } v = x : l \text{ and } x \text{ is a group variable} \end{cases}$$

Example 5.6. Following the above rewriting rules, query Q_{high} in Example 5.2 can be rewritten into Q^*_{high} as follows:

```
MATCH f0 =(a : Person|Company)-[]->(s : Wallet)-[f1 : Transfer]->(r)
FILTER a.riskScore >=0.7 AND s.balance < r.balance
RETURN a.name AS name, a.city AS city, [A(a) FOR a IN f0] AS t0,
  A(a : Person) AS t1, A(a : Company) AS t2, A(s : Wallet) AS t3,
  A(f1 : Transfer) AS t4, A(a.name) AS t5, A(a.city) AS t6,
  A(a.riskScore) AS t7, A(s.balance) AS t8, A(r.balance) AS t9
```

5.3 Decoding the Provenance Encodings

In the final phase of our provenance computation approach (Phase 3 of Algorithm 1), we decode the provenance encoding given by the output variables of the rewritten query. This allows us to reconstruct the annotations corresponding to the why-provenance $\mathcal{A}(\mathsf{WHY}(Q, G, \mu))$ for each answer tuple μ resulting from executing Q on G. Algorithm 4 describes the process of decoding, and takes as input the rewritten query Q^*, a graph G, and the set of output variables of Q, $\mathcal{V}_{out}$. First, it executes the rewritten query Q^* and initializes the result set (lines 1–2). Then, for each answer μ^* of executing Q^* on G, we extract the encoded provenance. Before we describe the precise decoding mechanism we need to highlight two properties of any given rewritten query answer μ^*. First, since the rewriting process retains the original query output variables in the

Algorithm 4 decodeProvenance($Q^*, G, \mathcal{V}_{out}$)

Require: Rewritten query Q^*, graph G, set of result variables $\mathcal{V}_{out}$
Ensure: Why-provenance annotated query results *results*
1: $[\![Q^*]\!]_G \leftarrow$ evaluateQuery(Q^*, G)
2: *results* $\leftarrow \emptyset$
3: **for all** tuple $\mu^* \in [\![Q^*]\!]_G$ **do**
4: Let μ be μ^* restricted to $\mathcal{V}_{out}$; $Wit \leftarrow \emptyset$
5: **for all** variables p in $attr(\mu^*) - \mathcal{V}_{out}$ **do**
6: $Wit \leftarrow Wit \cup \{\mu^*(p)\}$
7: $Wit \leftarrow flatten(Wit)$
8: **if** μ in *results* **then**
9: **if** Wit is coarse-minimal in μ **then**
10: Remove non-coarse minimal from μ wrt. Wit
11: $results[\mu] \leftarrow results[\mu] \cup \{Wit\}$
12: **else**
13: $results[\mu] \leftarrow \{Wit\}$
14: **return** *results*

rewritten query ($\mathcal{V}_{out}$), we can recover the original query answer tuple by restricting μ^* to $\mathcal{V}_{out}$. Second, since we have introduced all possible path variables (cf. Algorithm 2, line 7) and since witnesses are defined based on paths (cf. Definition 4.9), we know every μ^* encodes a witness for μ^* restricted to $\mathcal{V}_{out}$. In lines 4, we recover the original query answer μ and initialize its witness Wit. Then, we iterate over the provenance variables p (i.e., the output variables of μ^* that are not in $\mathcal{V}_{out}$) and collect them. Since the provenance variables can contain group and list variables, we flatten the witness after collection (lines 5–7). Finally, we store the witness Wit for μ (lines 8–13). If we want minimal witnesses (cf. Definition 4.9, condition (ii)) we need to check whether Wit is coarse-minimal with respect to the already computed witnesses $results[\mu]$ (lines 9–10). Since this is a computationally expensive step, one can choose to leave it out, replacing lines 9–11 with simply line 11.

Example 5.7. We revisit query Q_{high} (Example 5.2) and its rewritten version Q^*_{high} (Example 5.6) to demonstrate the decoding process. Evaluating Q^*_{high} over G_{crypto} yields the provenance encoding shown in Table 1 (Example 5.4). To compute the why-provenance for the result tuple $\mu' = \{name \mapsto Alice; city \mapsto Paris\}$, we decode the corresponding tuples from the rewritten result, μ^*_1 and μ^*_2. For μ^*_1, we take the flattened union of its provenance variables, constructing the witness $W_{\mu^*_1} = \{n_1, n_4, n_5, e_1, e_5, l_1, l_4, l_{12}, k_1, k_2, k_3, k_{11}, k_{13}\}$. This witness is then stored for tuple μ'. Repeating this for μ^*_2 yields $W_{\mu^*_2}$. Together, these two tuples give us the why-provenance $\mathcal{A}(\mathsf{WHY}(Q_{high}, G_{crypto}, \mu'))$.

6 EXPERIMENTS

In our experiments, we use standard benchmarks and evaluate our approach using Neo4j.

6.1 Experimental Setup

Environment. We implemented our approach as user-defined procedures for provenance computation in Neo4j using its Java API (Neo4j version 2025.09.0) and Java Development Kit (JDK) version 21. All experiments were conducted using Neo4j Community Edition (2025.09.0) [24] on a Debian 12 host equipped with an AMD

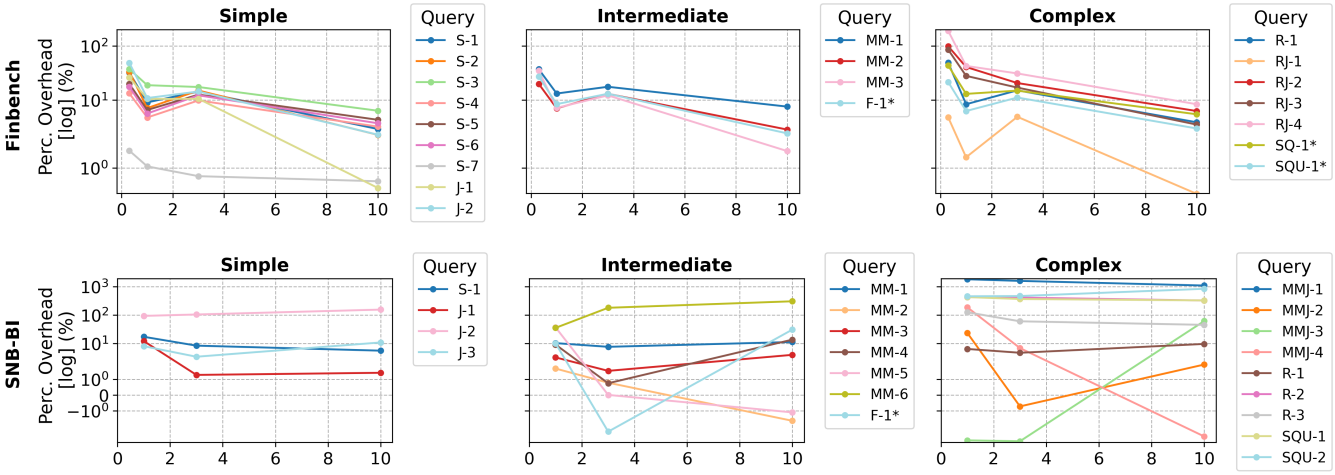


Figure 2: Provenance overhead vs. scale factor on FinBench (top) and SNB-BI (bottom)

EPYC 9254 24-core processor, 768 GB of RAM, and a 10 TB HDD. Prior to evaluation, the database cache was warmed. To the best of our knowledge, no existing system supports provenance computation for GQL queries over property graph databases. We evaluated the overhead induced by our approach by comparing the average execution times of our complete provenance computation process with those of the corresponding original (non-provenance) queries.

Datasets. We used two benchmark datasets from the Linked Data Benchmark Council (LDBC): the Financial Benchmark (FinBench) [25] and the Social Network Benchmark – Business Intelligence (SNB-BI) Benchmark. We employed the published scale factors 0.3, 1, 3, and 10 for FinBench [25], and scale factors 1, 3, and 10 for SNB-BI [28]. The characteristics of these datasets are summarized in Table 2.

Workloads. We used 17 and 19 query templates for FinBench and SNB-BI benchmarks respectively; this includes all benchmark templates except those involving updates. Aggregations, patterns in conditions, and certain expressions, such as *case*, *optional* and *with* were omitted from the templates since their particularities and non-monotonic nature go beyond the scope of this paper. Additionally, we introduced 3 new query templates for FinBench and 1 for SNB-BI to evaluate features not covered by the original benchmarks.

We characterize each query template based on its key features: pattern joins (**J**), i.e. PMQs of form $Q \bowtie Q'$, multiple `MATCH` clauses (**MM**), `FILTER` clauses (**F**), pattern repetitions (**R**), i.e. PMEs of form $\psi^{n..m}$, subqueries with `CALL` clauses (**SQ**), and union (**U**). For instance, a query template labeled **RJ-1** contains pattern repetitions and pattern joins, with the number serving as a unique identifier. Templates without any of these key features are labeled **S**, and those created by us are marked with an asterisk (*). For ease of presentation, we group the queries based on their runtime into three categories as *simple*, *intermediate*, and *complex*.

dataset	sf	# nodes	# edges	# properties
finbench	0.3	166,804	929,647	4,077,973
	1	556,082	3,101,291	13,602,020
	3	1,665,335	9,255,337	40,627,370
	10	5,376,981	26,633,151	117,049,301
snb-bi	1	2,997,352	17,196,776	35,076,475
	3	8,513,157	51,035,227	102,145,874
	10	27,231,349	170,343,945	334,465,133

Table 2: Characteristics of the datasets

Each query template is run with 20 parameter sets, each executed three times, and the average response time is reported. We use the published parameters for all dataset–scale factor combinations, except FinBench 0.3 and 3, where parameters were unpublished and thus were generated with the official generator[3]. To avoid bias from execution order, queries were issued in a randomized sequence.

6.2 Results

To quantify the cost of provenance computation, we report *percentage provenance overhead*, defined as the execution time difference between the original query and our implementation, including provenance computation, expressed as a percentage of the original query runtime. Figure 2 shows the percentage provenance overhead for each query across scale factors (logarithmic scale) for both FinBench and SNB-BI benchmarks.

For FinBench (Figure 2, top), the relative overhead rapidly decreases with increasing scale factor, where as it remains generally stable for SNB-BI (Figure 2, bottom). While for FinBench the relative overhead varies more across queries at low scale factors, it converges to a consistently low range at higher scale factors across all query categories. This indicates that for most queries, provenance computation becomes relatively cheaper or remains stable compared to baseline query execution as data size grows.

The relative overheads for queries **J-2**, **MM-6**, **MMJ-1**, **SQU-1**, and **SQU-2** in SNB-BI have the highest relative overheads across all scale factors of SNB-BI (Figure 2, bottom). These queries feature substantially longer pattern matching expressions (PMEs), more match clauses that trigger intermediate joins, and a large number

[3] https://github.com/ldbc/ldbc_finbench_datagen (Last accessed: March 1, 2026)

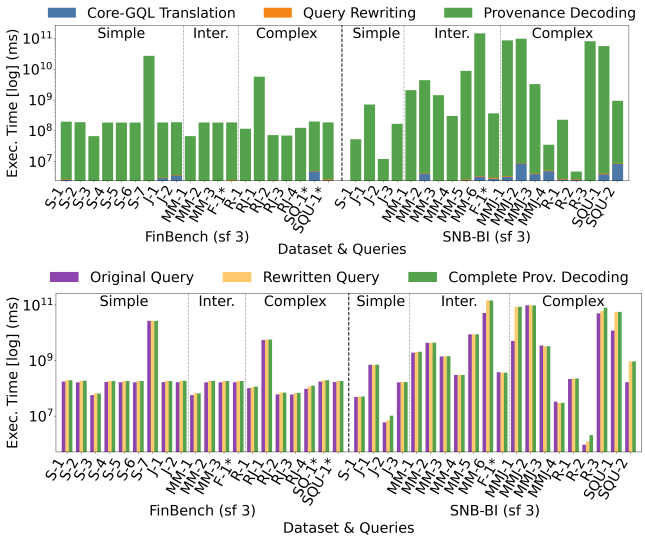


Figure 3: Cost breakdown of provenance computation per phase (top). Comparison of rewritten query execution against complete provenance decoding and original query execution (bottom). Executed on FinBench and SNB-BI (sf 3).

of intermediate results. As a result, the propagated provenance attributes in the rewritten queries also require additional joins, contributing to the higher overhead.

In addition, in Figure 2 (bottom) we observe negative overheads for queries **F-1***, **MM-2**, **MM-5**, **MMJ-2**, **MMJ-3**, and **MMJ-4** in SNB-BI and an unexpected drop in the percentage overhead for **J-3** in SNB-BI. This phenomenon can be attributed to the query optimizer selecting a significantly different execution plan for the rewritten provenance query compared for the original query.

We next examine the cost of the three phases of Algorithm 1 in detail. Figure 3 (top) reports the average execution time of the individual phases for every query and dataset. The combined cost of the translation and query rewriting (phases 1–2) remains small and stable across queries, with its variation reflecting differences in the query complexities and the number of used labels and properties.

In contrast, provenance decoding (phase 3) dominates the total cost in most cases. Queries **S-7** and **RJ-1** from FinBench, as well as **MM-6**, **MMJ-1**, **R-3**, and **SQU-1** from SNB-BI, are notably more expensive which can be attributed to their large result sizes. This shows that translation and rewriting contribute only a small fixed cost; the cost of provenance computation is primarily driven by the decoding phase.

Since phase 3 is the most significant, we investigate its overhead in more depth. It includes the execution of the rewritten query and combines this with the decoding itself. In our implementation, we decode in a streaming manner: as query results come in, they are decoded and possibly added to the output witness sets. As a consequence, the cost of query execution cannot be isolated from the decoding. Nevertheless, to show that most time is spent on executing the rewritten query, we report the rewritten query execution time, compared to the time it takes to execute phase 3 in total.

Figure 3 (bottom) shows three execution times: of the rewritten query, of phase 3, and of the original query for reference. Across all

sf	0.3	1	3	10
finbench	87%	82%	86%	81%
snb-bi	-	81%	85%	86%

Table 3: Average reduction in witness size for fine-grained provenance compared to coarse provenance.

queries, execution of the rewritten query dominates. This indicates that provenance computation is largely bound by the rewritten query execution and not the decoding itself. Furthermore, as Figure 3 (bottom) shows, the overhead introduced by the rewritten query relative to the original query remains modest. This shows that, overall, the provenance computation itself is lightweight.

Finally, we evaluate the conciseness of fine-grained provenance compared to coarse provenance. Recall that coarse provenance includes *full* nodes and edges, while fine-grained provenance includes only the necessary labels and properties, resulting in smaller witnesses. We report the percentage reduction in witness size, defined as the difference between the coarse and fine-grained witness size expressed as a percentage of the coarse witness size. Table 3 reports the average percentage reduction in witness size across all the witnesses of the query results, for each dataset and scale factor. The reductions range from 80% to 90%, demonstrating that fine-grained provenance consistently produces substantially smaller witnesses than coarse-grained provenance across all configurations. This large reduction can be explained by the richness of the underlying property graph schema as fine-grained provenance retains only the labels and properties relevant to a query result while excluding irrelevant data. These results underscore the effectiveness of fine-grained provenance in generating more concise explanations.

7 CONCLUSION

Property graphs have become a widely adopted model for representing complex, interconnected data, and the emergence of GQL provides a standardized foundation for querying them. Understanding the origin of query results is essential for ensuring transparency and reliability, requiring principled mechanisms to determine which parts of the graph are responsible for a specific answer. In this paper, we addressed this need by formally defining why-provenance for property graph queries building upon GQL and its algebraic core. Our approach precisely captures the graph elements contributing to the derivation of query results. To support practical applicability, we developed a query-rewriting technique that enables provenance computation directly within existing GQL-enabled systems. The experimental evaluation with Neo4j shows that the additional computational costs remain manageable and scale with both data size and query complexity. As future work, we plan to extend our framework to additional language constructs and, in particular, to non-monotonic operators. As known from research in the relational setting [6], such operators introduce fundamental challenges for provenance computation, and addressing them in the context of property graphs remains an important direction for the future.

ACKNOWLEDGMENTS

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013].

REFERENCES

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

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

[3] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bull.* 41, 1 (2018), 51–62.

[4] Zubaria Asma, Daniel Hernández, Luis Galárraga, Giorgos Flouris, Irini Fundulaki, and Katja Hose. 2024. NPCS: Native Provenance Computation for SPARQL. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*. 2085–2093.

[5] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*. 316–330.

[6] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (2009), 379–474.

[7] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. 2012. Provenance for SPARQL Queries. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*. 625–640.

[8] Alin Deutsch, Nadime Francis, Alastair Green, Keith W. Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 2246–2258.

[9] Benjamín Farias, Wim Martens, Carlos Rojas, and Domagoj Vrgoc. 2024. PathFinder: Returning Paths in Graph Queries. In *The Semantic Web - ISWC 2024 - 23rd International Semantic Web Conference, Baltimore, MD, USA, November 11-15, 2024, Proceedings, Part II*. 135–154.

[10] 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.

[11] 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, 1:1–1:22.

[12] Floris Geerts and Antonella Poggi. 2010. On database query languages for K-relations. *J. Appl. Log.* 8, 2 (2010), 173–185.

[13] Floris Geerts, Thomas Unger, Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. 2016. Algebraic Structures for Capturing the Provenance of SPARQL Queries. *J. ACM* 63, 1 (2016), 7:1–7:63.

[14] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proc. VLDB Endow.* 18, 6 (2025), 1798–1810.

[15] Boris Glavic. 2021. Data Provenance. *Found. Trends Databases* 9, 3-4 (2021), 209–441.

[16] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. 174–185.

[17] Gösta Grahne, Tianyi Liu, and Nematollaah Shiri. 2022. Universal provenance for regular path queries. In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance, TaPP 2022, Philadelphia, Pennsylvania, 17 June 2022*. 4:1–4:7.

[18] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. ACM, 31–40.

[19] Daniel Hernández, Luis Galárraga, and Katja Hose. 2021. Computing How-Provenance for SPARQL Queries via Query Rewriting. *Proc. VLDB Endow.* 14, 13 (2021), 3389–3401.

[20] Robert Ikeda, Junsang Cho, Charlie Fang, Semih Salihoglu, Satoshi Torikai, and Jennifer Widom. 2012. Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. 1249–1252.

[21] International Organization for Standardization. 2024. *Information technology — Database languages — GQL*. Standard ISO/IEC CD 39075:2024. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/76120.html

[22] Leonid Libkin, Wim Martens, Filip Murlak, Liat Peterfreund, and Domagoj Vrgoc. 2025. Querying Graph Data: Where We Are and Where To Go. In *Companion of the 44th Symposium on Principles of Database Systems, PODS 2025, Berlin, Germany, June 22-27, 2025*. 9–26.

[23] Tobias Müller and Pascal Engel. 2022. How, Where, and Why Data Provenance Improves Query Debugging: A Visual Demonstration of Fine-Grained Provenance Analysis for SQL. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. 3178–3181.

[24] Neo4j, Inc. 2025. *Neo4j Graph Database*. Neo4j, Inc. https://neo4j.com/ Released 29 September 2025; Graph Database Management System.

[25] Shipeng Qi, Bing Tong, Jiatao Hu, Heng Lin, Yue Pang, Wei Yuan, Songlin Lyu, Zhihui Guo, Ke Huang, Xujin Ba, Qiang Yin, Youren Shen, Yan Zhou, Tao Lv, Jia Li, Lei Zou, Yongwei Wu, Gábor Szárnyas, Xiaowei Zhu, Wenguang Chen, and Chuntao Hong. 2025. The LDBC Financial Benchmark: Transaction Workload. *Proc. VLDB Endow.* 18, 9 (2025), 3007–3020.

[26] Yann Ramusat, Silviu Maniu, and Pierre Senellart. 2021. Provenance-Based Algorithms for Rich Queries over Graph Databases. In *EDBT*. OpenProceedings.org, 73–84.

[27] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proc. VLDB Endow.* 11, 12 (2018), 2034–2037.

[28] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. 2018. An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES-NDA at SIGMOD/PODS*. ACM, 9:1–9:11.

[29] Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. 2015. A Demonstration of TripleProv: Tracking and Querying Provenance over Web Data. *Proc. VLDB Endow.* 8, 12 (2015), 1992–1995.