On the Expressiveness of Languages for Querying Property Graphs in Relational Databases

HADAR ROTSCHIELD, The Hebrew University of Jerusalem, Israel
LIAT PETERFREUND, The Hebrew University of Jerusalem, Israel

SQL/PGQ is the emerging ISO standard for querying property graphs defined as views over relational data. We formalize its expressive power across three fragments: the read-only core, the read-write extension, and an extended variant with richer view definitions. Our results show that graph creation plays a central role in determining the expressiveness.

The read-only fragment is strictly weaker than the read-write fragment, and the latter is still below the complexity class NL. Extending view definitions with arbitrary arity identifiers closes this gap: the extended fragment captures exactly NL. This yields a strict hierarchy of SQL/PGQ fragments, whose union covers all NL queries. On ordered structures the hierarchy collapses: once arity-2 identifiers are allowed, higher arities add no power, mirroring the classical transitive-closure collapse and underscoring the central role of view construction in property graph querying.

CCS Concepts: • **Information systems → Graph-based database models**; **Query languages**; **Data management systems**; • **Theory of computation → Logic and databases**; **Pattern matching**.

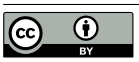
Additional Key Words and Phrases: Graph Query Languages, GQL Standard, SQL/PGQ, Property Graph Model, Expressive Power

ACM Reference Format:
Hadar Rotschield and Liat Peterfreund. 2025. On the Expressiveness of Languages for Querying Property Graphs in Relational Databases . *Proc. ACM Manag. Data* 3, 5 (PODS), Article 279 (November 2025), 18 pages. https://doi.org/10.1145/3767715

1 Introduction

Property graphs are now widely used in industry, supporting key applications in analytics, fraud detection, and recommendation systems [18, 21]. Property graphs have evolved from simpler graph data models such as edge-labeled graphs and RDF triples, offering a richer and more flexible structure [14]. In a property graph, both nodes and edges can be annotated with labels that can be thought of as types, and can carry arbitrary sets of key-value pairs as properties [2]. This design allows property graphs to model heterogeneous and complex domains naturally. Driven by the wide adoption of property graphs in real-world applications, the International Organization for Standardization (ISO), together with partners from academia and industry, is developing a new standard: GQL [13]. The standard has two components: a standalone graph query language GQL, and SQL/PGQ, which integrates graph querying into relational databases. As these standards take shape, there is a growing need for solid theoretical foundations to guide their development and support the design of robust, graph-aware relational engines.

Authors' Contact Information: Hadar Rotschield, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel, hadar.rotschield@mail.huji.ac.il; Liat Peterfreund, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel, liat.peterfreund@mail.huji.ac.il.



ACM 2836-6573/2025/11-ART279
https://doi.org/10.1145/3767715

In a similar spirit to [22], we observe that recent developments in graph data management follow a familiar historical trajectory: instead of replacing the relational model, new paradigms like property graphs are being integrated into it. SQL continues to evolve by incorporating core ideas from alternative models, now through extensions like SQL/PGQ, supporting property graph querying natively within the relational framework.

Motivated by this ongoing convergence of graph and relational models, we take a step towards a deeper understanding of SQL/PGQ by formally analyzing its expressive power and identifying the key factors that influence it.

SQL/PGQ and Property Graph Views. SQL/PGQ is an extension of SQL for querying property graphs defined as relational views. The language operates across three conceptual layers: (i) a pattern matching layer, where graph patterns are matched against the view-defined graph resulting in relations; (ii) a relational algebra layer, which manipulates the results of pattern matching using standard SQL constructs; and (iii) a view creation layer, where the actual graph structure, that is, nodes, edges, labels, and properties, is constructed. While layers (i) and (ii) have received attention in previous works, the view creation step (iii) has remained under-explored.

The role of view creation in the read-write fragment of SQL/PGQ is significant. Intermediate relational results may themselves be used to generate new graphs, enabling recursive-like compositions that entangle all three layers. Our key observation is that the expressive power of SQL/PGQ is ultimately governed by the flexibility given in layer (iii), that is, its graph view definitions. Understanding and formally capturing this mechanism is essential for characterizing the expressiveness.

Example 1.1 (Graph View: Transfers Between Bank Accounts). Consider a relational schema that consists of the relations

- `Account(iban)` that lists all bank accounts by their unique IBAN, and
- `Transfer(t_id, src_iban, tgt_iban, ts, amount)` that records transfers identified uniquely by `t_id` from `src_iban` to `tgt_iban` at time `ts`, with amount `amount`.

To analyze suspicious sequences of transfers, we can define the following SQL/PGQ graph view:

```
CREATE PROPERTY GRAPH Transfers (
  NODES TABLE
    Account KEY (iban) LABEL Account,
  EDGES TABLE
    Transfer KEY (t_id)
        SOURCE KEY src_iban REFERENCES Account
        TARGET KEY tgt_iban REFERENCES Account
        LABELS Transfer PROPERTIES (ts, amount) );
```

This creates a (directed) property graph in which nodes are accounts (identified by `iban`), and edges represent transfers (identified by `t_id`) from sender to recipient. Edges are labeled and store `ts` and `amount` as properties. On this property graph SQL/PGQ applies pattern matching.

Problem Statement. In this work, we take a step toward understanding the expressive power of SQL/PGQ by focusing on the role of graph view definitions. Prior abstractions of SQL/PGQ were proposed [9, 10], with the most recent effort [12] extracting the core of the read-only fragment but overlooking the graph view construction layer. Our first goal is to provide a precise formalization of the read-only fragment, PGQ^{RO}, consistent with the standard [16], and to fill this gap in layer (iii) by making explicit how graphs are derived from relational data. Building on this formalization, we compare the expressive power of the read-only PGQ^{RO} and the read-write PGQ^{RW} fragments. We show that while the read-write variant PGQ^{RW} allows for graph views to be constructed and

queried dynamically, enabling more expressive compositions, it still does not capture all queries in NL, a natural benchmark for expressive power and complexity for graph query languages.

To explain this gap, we identify key features that remain missing from SQL/PGQ's current design, most notably the lack of tuple-based identifiers and richer view constructs. By addressing these limitations, we define an extended read-write fragment, PGQ^{EXT}, that supports more powerful graph constructions. This leads to a refined classification of the expressive capabilities of SQL/PGQ as we formally characterize the boundaries between PGQ^{RO}, PGQ^{RW}, and PGQ^{EXT}.

Our Contributions. This work provides a precise characterization of the expressive power of SQL/PGQ by isolating the role of graph view definitions. Specifically:

(1) **Semantics of SQL/PGQ fragments.** We define the formal syntax and semantics of three fragments: the read-only fragment PGQ^{RO}, the read-write fragment PGQ^{RW}, and the extended fragment PGQ^{EXT}, which supports graph views with arbitrary n-ary node and edge identifiers.
(2) **Capturing NL with view-based recursion.** We show that PGQ^{EXT} captures exactly the expressive power of FO[TC] (first-order logic with transitive closure), and thus all of NL on ordered structures. In contrast, PGQ^{RW} remains strictly below this bound, establishing the separation $\text{PGQ}^{\text{RO}} \subsetneq \text{PGQ}^{\text{RW}} \subsetneq \text{PGQ}^{\text{EXT}} = \text{FO[TC]} = \text{NL}$.
(3) **Arity-based expressiveness.** We identify a hierarchy within PGQ^{EXT} based on the maximum arity n of node and edge identifiers. Each level PGQ^{n} corresponds exactly to $\text{FO[TC}^{n}\text{]}$, the fragment of first-order logic with n-ary transitive closure. The hierarchy is strict up to arity 2, where it collapses (on ordered structures): higher arities add no expressive power. Thus, arity-2 identifiers suffice to capture the full power of PGQ^{EXT}.
(4) **Descriptive complexity.** We establish tight data complexity bounds for all fragments, situating SQL/PGQ's expressive power within standard complexity classes and connecting it to descriptive complexity theory.

Technical Preview. We link SQL/PGQ to first-order logic with transitive closure (FO[TC]) through constructive, bidirectional translations in the spirit of [8]. This lets us transfer results both ways and isolate what each fragment can express. Using these translations, we exhibit concrete separator queries: each lies in the stronger fragment but is provably absent from the weaker, pinpointing the impact of view creation and n-ary identifiers on recursion.

Related Work. Early foundational work on graph querying centered on *regular path queries* (RPQs) and their conjunctive and two-way extensions [3, 4, 7]. Nested regular expressions further enrich RPQs with controlled nesting and synchronization [5], and regular queries offer a non-recursive Datalog layer and capture NL [20]. These formalisms assume *edge-labeled graphs* without node or edge properties, unlike the property graph model adopted by, e.g., Cypher, G-CORE, and PGQL [1, 19, 23].

Beyond these classical foundations, recent efforts have sought to better align theoretical formalisms with the richer features of practical languages. In particular, dl-CRPQs extend conjunctive regular path queries with node and edge treatment, list variables, path modes, and data filters, thereby mirroring key design choices in Cypher, SQL/PGQ, and GQL [17]. This work situates modern languages within automata-theoretic foundations and highlights opportunities for optimization.

Our work closes a complementary key gap by isolating two features missing from earlier formal models, view creation and n-ary identifiers, and showing that their addition lifts SQL/PGQ to the full power of FO[TC]. This means PGQ^{EXT} captures exactly the NL queries, positioning it firmly within a well-studied expressiveness class. Importantly, this is achieved without relying on general fixed-point logic, keeping evaluation complexity strictly below P. In contrast to earlier results

focused on edge-labeled graphs, our analysis establishes NL expressiveness in the richer settings of property graphs.

Paper Organization. Section 2 covers background on property graphs and pattern matching. Section 3 defines the read-only fragment PGQ^{RO}, and Section 4 introduces the read-write fragment PGQ^{RW} and compares its expressiveness to PGQ^{RO}. Section 5 defines the extended fragment PGQ^{EXT}, and Section 6 characterizes its expressive power. Section 7 discusses deviations of our formalization from the standard, and Section 8 concludes with open research directions. Full proofs and additional details are in the Appendix of the full version of the paper.

2 Preliminaries

We review the basic definitions of property graphs and relational structures, and describe the pattern matching layer that extracts relations from property graphs.

2.1 Property Graphs and Relational Structures

Property Graphs. We use the standard definition of property graphs [9]. Assume pairwise disjoint countable sets $\mathcal{L}$ of labels, $\mathcal{K}$ of keys, $\mathcal{P}$ of properties, $\mathcal{N}$ of node IDs, and $\mathcal{E}$ of edge IDs.

Definition 2.1 (Property Graphs)**.** A property graph is a tuple $G = \langle N, E, \text{src}, \text{tgt}, \text{lab}, \text{prop}\rangle$ where

- $N \subset \mathcal{N}$ is a finite set of node IDs used in G;
- $E \subset \mathcal{E}$ is a finite set of directed edge IDs used in G;
- $\text{src}, \text{tgt} : E \to N$ define source and target of an edge;
- $\text{lab} : N \cup E \to 2^{\mathcal{L}}$ is a labeling function that associates with every node or edge ID a (possibly empty) finite set of labels from $\mathcal{L}$;
- $\text{prop} : (N \cup E) \times \mathcal{K} \rightharpoonup \mathcal{P}$ is a finite partial function that associates a property value with a node/edge id and a key.

Relational Structures. We assume two infinite sets: the set $\mathcal{R}$ of relation names R, and the set $\mathcal{C}$ of domain elements. We follow the unnamed perspective, and associate each relation name $R \in \mathcal{R}$ with a positive integer $\text{arity}(R)$. A *(database) schema* $\mathcal{S} \subseteq \mathcal{R}$ is a finite set of relation names. A *relation* $\mathbf{R}$ over a relation name R is a finite set of *tuples* in $\mathcal{C}^{\text{arity}(R)}$. A *database (instance)* $\mathcal{D}$ over a schema $\mathcal{S}$ assigns to each relation name $R \in \mathcal{S}$ a *relation* $R^{\mathcal{D}}$ over R. The active domain of $\text{adom}(\mathcal{D})$ of $\mathcal{D}$ consists of all constants that appear in $\mathcal{D}$.

Remark 2.1. Throughout the paper, we assume our structures are ordered, reflecting systems where order is implicit and aligning with standard practice in database theory.

Relations as (partial) functions. A relation R_f is said to *encode a function* $f : X \to Y$ if $\text{arity}(R_f) = \text{arity}(X) + \text{arity}(Y)$, and for every $x \in X$, there exists a unique $y \in Y$ such that $(x, y) \in R_f$. Similarly, R_f encodes a *partial function* $f : X \rightharpoonup Y$ if the same condition on the arities holds, and for every $x \in X$, there exists at most one $y \in Y$ such that $(x, y) \in R_f$.

2.2 Pattern Matching: From Property Graphs to Relations

Pattern matching is the key component of graph query languages as it extracts relations from property graphs. We use the refined definitions of core PGQ from [12]. We fix an infinite set Vars of variables and define *patterns* ψ and *output patterns* ψ_Ω in Figure 1.

2.3 Pattern Matching Semantics

In Figure 2, we present core PGQ pattern matching semantics as originally introduced in [12], and later further simplified in [8].

Patterns:

$$\psi, \psi_1, \psi_2 \;:=\; (x) \mid \overset{x}{\rightarrow} \mid \overset{x}{\leftarrow} \mid \psi_1\,\psi_2 \mid \psi^{n..m} \mid \psi\langle\theta\rangle \mid \psi_1 + \psi_2 \text{ if } \mathsf{fv}(\psi_1) = \mathsf{fv}(\psi_2)$$

where $x \in \mathsf{Vars}$ is optional, and $0 \le n \le m \le \infty$.

Conditions:

$$\theta, \theta' \;:=\; x.k = x'.k' \mid \ell(x) \mid \theta \vee \theta' \mid \theta \wedge \theta' \mid \neg\theta \quad \text{where } x, x' \in \mathsf{Vars},\ k, k' \in \mathcal{K}.$$

Free Variables:

$$\mathsf{fv}\Big((x)\Big) = \mathsf{fv}\left(\overset{x}{\rightarrow}\right) = \mathsf{fv}\left(\overset{x}{\leftarrow}\right) := \{x\}; \quad \mathsf{fv}(\psi_1\,\psi_2) := \mathsf{fv}(\psi_1) \cup \mathsf{fv}(\psi_2);$$
$$\mathsf{fv}(\psi^{n..m}) := \emptyset; \quad \mathsf{fv}(\psi\langle\theta\rangle) := \mathsf{fv}(\psi); \quad \mathsf{fv}(\psi_1 + \psi_2) := \mathsf{fv}(\psi_1).$$

Output Patterns:

$$\psi_\Omega; \quad \mathsf{fv}(\psi_\Omega) = \{\omega_1, \ldots, \omega_n\}$$

where $\Omega := (\omega_1, \ldots, \omega_n), n \ge 0,\ \omega_i \ne \omega_j$ for $i \ne j,\ \omega_i \in \mathsf{Vars} \cup \{x.k \mid x \in \mathsf{Vars},\ k \in \mathcal{K}\}$.

Fig. 1. Syntax of Patterns and Output Patterns.

2.3.1 Patterns. The semantics $[\![\psi]\!]_G$ of a pattern ψ on G is defined as a set of triples (s, t, μ), where s and t are the source and target nodes of a path matching ψ, and μ is a *variable mapping* assigning matched graph elements (that is, nodes and edges) to the pattern's free variables.[1]

The semantics uses the following definitions:

Operations on variable mappings. We use standard operations on mappings in the semantics. Given a mapping μ and a subset $X \subseteq \mathsf{dom}(\mu)$, the restriction $\mu \upharpoonright X$ is the mapping with domain X such that $(\mu \upharpoonright X)(x) := \mu(x)$ for all $x \in X$. We write $\mu_\emptyset$ to denote the mapping with an empty domain.

Two mappings μ_1 and μ_2 are *compatible*, written $\mu_1 \sim \mu_2$, if they agree on all common variables. Their *union*, written $\mu_1 \bowtie \mu_2$, is the mapping defined over $\mathsf{dom}(\mu_1) \cup \mathsf{dom}(\mu_2)$, with values taken from μ_1 and μ_2 accordingly.

Condition Satisfaction. A mapping μ satisfies a condition θ, written $\mu \models \theta$, as follows:

- $\mu \models x.k = x'.k'$ if both $\mathsf{prop}(\mu(x), k)$ and $\mathsf{prop}(\mu(x'), k')$ are defined and equal;
- $\mu \models \ell(x)$ if $\ell \in \mathsf{lab}(\mu(x))$.

Satisfaction extends to Boolean combinations of conditions with the standard semantics of $\wedge$, $\vee$, $\neg$.

2.3.2 Output Patterns. The semantics $[\![\psi_\Omega]\!]_G$ of output patterns ψ_Ω on G is a set of tuples:

$$(\mu_\Omega(\omega_1), \ldots, \mu_\Omega(\omega_n)),$$

where each $\mu_\Omega(\omega_i)$ is either a node identifier, an edge identifier, or a property value. This set is finite because μ_Ω ranges over mappings with finite domain and finite codomain.

To ensure compatibility with the relational model introduced in Section 2, we assume without loss of generality that $\mathcal{N} \cup \mathcal{E} \cup \mathcal{P} \subseteq \mathcal{C}$. That is, all output values belong to the relational domain $\mathcal{C}$, so that the result of evaluating an output pattern can be interpreted as a relation. This aligns with the relational layer of PGQ, which allows further use of relational algebra on the output of pattern matching.

[1]Our simplification is based on the fact that the semantics avoids storing the full path, and instead it preserves the key information needed for composing patterns, that is, its source and target.

Patterns:

$$
\begin{aligned}
[\![(x)]\!]_G &:= \{\, (n, n, \{x \mapsto n\}) \mid n \in N \,\} \\
\left[\!\!\left[\overset{x}{\rightarrow} \right]\!\!\right]_G &:= \{\, (n_1, n_2, \{x \mapsto e\}) \mid e \in E,\ \mathrm{src}(e) = n_1,\ \mathrm{tgt}(e) = n_2 \,\} \\
\left[\!\!\left[\overset{x}{\leftarrow} \right]\!\!\right]_G &:= \{\, (n_2, n_1, \{x \mapsto e\}) \mid e \in E,\ \mathrm{src}(e) = n_1,\ \mathrm{tgt}(e) = n_2 \,\} \\
[\![\psi_1 + \psi_2]\!]_G &:= [\![\psi_1]\!]_G \cup [\![\psi_2]\!]_G \\
[\![\psi_1\, \psi_2]\!]_G &:= \{\, (s_1, t_2, \mu_1 \bowtie \mu_2) \mid (s_1, n, \mu_1) \in [\![\psi_1]\!]_G,\ (n, t_2, \mu_2) \in [\![\psi_2]\!]_G, \mu_1 \sim \mu_2 \,\} \\
[\![\psi\langle\theta\rangle]\!]_G &:= \{\, (s, t, \mu) \in [\![\psi]\!]_G \mid \mu \models \theta \,\} \\
[\![\psi^{n..m}]\!]_G &:= \bigcup_{i=n}^{m} [\![\psi]\!]_G^i \quad \text{where} \quad [\![\psi]\!]_G^0 := \{\, (n, n, \mu_\emptyset) \mid n \in N \,\}, \quad \text{and for } n > 0 \\
[\![\psi]\!]_G^n &:= \{\, (s_1, t_n, \mu_\emptyset) \mid \exists \mu_1, \ldots, \mu_n :\ (s_i, t_i, \mu_i) \in [\![\psi]\!]_G \text{ and } t_i = s_{i+1} \text{ for all } i < n \,\},
\end{aligned}
$$

Output Patterns:

$$
[\![\psi_\Omega]\!]_G := \{\, (\mu_\Omega(\omega_1), \ldots, \mu_\Omega(\omega_n)) \mid \exists s, t :\ (s, t, \mu) \in [\![\psi]\!]_G, \Omega := (\omega_1, \ldots, \omega_n) \,\}
$$

Fig. 2. Semantics of Patterns and Output Patterns.

Example 2.1 (Pattern Matching on Transfers Graph). Consider the property graph defined in Example 1.1. Suppose we want to find pairs of account IBANs (x, y) such that there exists a non-empty sequence of transfers from x to y where each transfer in this sequence has an amount greater than 100. To this end, we use the following output pattern:

$$
\Big((x) \overset{t}{\rightarrow}^{1,\infty} (y) \langle \theta \rangle \Big)_{x.\mathtt{iban}, y.\mathtt{iban}} \qquad \text{where} \quad \theta := \mathtt{Transfer}(t) \wedge t.\mathtt{amount} > 100
$$

which is written in SQL/PGQ as follows:

```
SELECT *
FROM GRAPH_TABLE (Transfers MATCH (x) -[t:Transfer]-> +(y)
                            WHERE t.amount > 100
                            RETURN (x.iban,y.iban) );
```

2.4 Expressiveness and Evaluation Complexity

Query Evaluation Problem. For a query language $\mathcal{L}$, the *evaluation problem* asks: given a query $Q \in \mathcal{L}$, a database $\mathcal{D}$, and a candidate tuple $\bar{a}$, decide whether $\bar{a}$ is in the result of evaluating Q on $\mathcal{D}$. We focus on the *data complexity* of this problem, that is, the query Q is fixed, and the complexity is measured with respect to the size of the input database $\mathcal{D}$.

Complexity Class NL. We use NL to denote the class of decision problems that can be solved by a nondeterministic Turing machine using $O(\log n)$ work tape, where n is the size of the input.

Language Expressiveness. We say that a query language $\mathcal{L}_2$ is *at least as expressive as* $\mathcal{L}_1$ if for every query $Q \in \mathcal{L}_1$ there exists a query $Q' \in \mathcal{L}_2$ such that $[\![Q]\!]_{\mathcal{D}} = [\![Q']\!]_{\mathcal{D}}$ on all databases $\mathcal{D}$. If, in addition, there exists a query in $\mathcal{L}_2$ not expressible in $\mathcal{L}_1$, then $\mathcal{L}_2$ is *strictly more expressive*, written $\mathcal{L}_1 \subsetneq \mathcal{L}_2$.

3 Read-Only PGQ

Building on the formalization of core PGQ presented in [12], we further develop the read-only fragment of the language and revisit certain aspects that were only briefly addressed in that work.

In particular, we focus on the foundational question: what property graph does the query operate on? While this issue was understandably not a central focus of GQL, which operates directly on property graphs, a more precise formalization is essential for PGQ, which is defined over relational databases and constructs property graph views dynamically.

3.1 Property Graph Views

We now describe the mechanism for constructing a *property graph view*, or *tabular property graph*, following the Standard's terminology, from a relational database. This view is only defined when the input database satisfies specific structural constraints, formalized below.

Definition 3.1. A *property graph view* is a relational database $\mathcal{D}$ over a schema consisting of six relation symbols, unary R_1, R_2, binary R_3, R_4, R_5, and a ternary R_6, for which the following hold:

(1) $R_1^{\mathcal{D}}$ and $R_2^{\mathcal{D}}$ are disjoint relations (representing node and edge identifiers, respectively).
(2) $R_3^{\mathcal{D}}, R_4^{\mathcal{D}}$ encode functions $R_2^{\mathcal{D}} \to R_1^{\mathcal{D}}$ (representing the source and target of each edge, respectively).
(3) $R_5^{\mathcal{D}} \subseteq (R_1^{\mathcal{D}} \cup R_2^{\mathcal{D}}) \times C$ (representing the labeling of nodes and edges).
(4) $R_6^{\mathcal{D}}$ encodes a partial function $(R_1^{\mathcal{D}} \cup R_2^{\mathcal{D}}) \times C \rightharpoonup C$ (representing properties).

Intuitively, $R_1^{\mathcal{D}}$ and $R_2^{\mathcal{D}}$ identify nodes and edges, $R_3^{\mathcal{D}}$ and $R_4^{\mathcal{D}}$ assign each edge a source and target node, and $R_5^{\mathcal{D}}$ and $R_6^{\mathcal{D}}$ capture labels and key-value properties of nodes and edges, respectively. Note that $R_5^{\mathcal{D}}$ and $R_6^{\mathcal{D}}$ may be empty, reflecting a scenario in which there are no labels or properties. The previous definition specifies when a relational database can be interpreted as a property graph. We now define the partial function pgView, which formalizes this interpretation.

Definition 3.2. We define pgView as the partial function that, given a sequence of relations $(\mathbf{R}_1, \ldots, \mathbf{R}_6)$ that satisfies the property graph view conditions, returns the property graph $\langle N, E, \mathsf{src}, \mathsf{tgt}, \mathsf{lab}, \mathsf{prop}\rangle$, denoted $\mathsf{pgView}(\mathbf{R}_1, \ldots, \mathbf{R}_6)$, where:

$$N := \mathbf{R}_1, \quad E := \mathbf{R}_2, \quad \mathsf{src} := \mathbf{R}_3, \quad \mathsf{tgt} := \mathbf{R}_4, \quad \mathsf{lab} := \mathbf{R}_5, \quad \mathsf{prop} := \mathbf{R}_6.$$

In PGQ, one applies pattern matching on such property graph views.

3.2 Read-Only PGQ: The Language

Output patterns return relations (finite sets of tuples), which in PGQ can be further manipulated using standard relational algebra. Since pattern matching operates over property graphs, we must interpret certain parts of the input relational database as defining such a graph. To formalize this, we define *read-only PGQ queries* namely $\mathrm{PGQ}^{\mathrm{RO}}$. A query $Q \in \mathrm{PGQ}^{\mathrm{RO}}$ over a relational schema $\mathcal{S}$ is defined in Figure 3, in the $\mathrm{PGQ}^{\mathrm{RO}}$ part.

Here, $\bar{R} = (R_1, \ldots, R_6)$ is a tuple of relation names from $\mathcal{S}$, defining the *property graph view* on which the output pattern ψ_Ω is evaluated. The semantics of the relational algebra operators follow their standard definitions; the novel aspect lies in the evaluation of the output pattern on the graph view. We define the semantics of a query Q over a database $\mathcal{D}$ as a relation $[\![Q]\!]_{\mathcal{D}}$, given in Figure 4 in the $\mathrm{PGQ}^{\mathrm{RO}}$ part.

Given a tuple $\bar{t} = (t_1, \ldots, t_n)$, we write $\bar{t} \models \theta$ to indicate that $\bar{t}$ satisfies the condition θ, with the following standard semantics:

$$\bar{t} \models \$i = \$i' \quad \text{iff} \quad 1 \le i, i' \le n \text{ and } t_i = t_{i'}.$$

This is extended to Boolean connectives $\wedge, \vee, \neg$ in the usual way. The function pgView is defined as in Definition 3.2.

$$
\begin{aligned}
&\textbf{PGQ}^{\textbf{RO}}\textbf{:}\\
&Q \;:=\; \psi_\Omega(\bar{R}) \mid R \mid \pi_{\$i_1,\ldots,\$i_k}(Q) \mid \sigma_\theta(Q) \mid Q \times Q' \mid Q \cup Q' \mid Q - Q'\\
&\theta \;:=\; \$i_1 = \$i_2 \mid \neg\theta \mid \theta \vee \theta \mid \theta \wedge \theta \quad \text{where } i_1,\ldots,i_k \in \mathbb{N}\\
&\textbf{PGQ}^{\textbf{RW}}\textbf{:}\\
&Q \;:=\; c \mid \psi_\Omega(\bar{Q})\\
&\textbf{PGQ}^{\textbf{EXT}}\textbf{:}\\
&Q \;:=\; \psi^{\text{ext}}_\Omega(\bar{Q})
\end{aligned}
$$

Fig. 3. Syntax of PGQ$^{\text{RO}}$, PGQ$^{\text{RW}}$, and PGQ$^{\text{EXT}}$: Each part shows the additions to the previous fragment: PGQ$^{\text{RW}}$ extends PGQ$^{\text{RO}}$ with constants and pattern matching on query results, and PGQ$^{\text{EXT}}$ further extends PGQ$^{\text{RW}}$ with pattern matching on more elaborate query results.

PGQ$^{\text{RO}}$ semantics:

$$
\begin{aligned}
[\![\psi_\Omega(\bar{R})]\!]_{\mathcal{D}} &:= [\![\psi_\Omega]\!]_G, \quad \text{where } G = \text{pgView}([\![R_1]\!]_{\mathcal{D}},\ldots,[\![R_6]\!]_{\mathcal{D}})\\
[\![R]\!]_{\mathcal{D}} &:= R^{\mathcal{D}}\\
[\![\pi_{\$i_1,\ldots,\$i_k}(Q)]\!]_{\mathcal{D}} &:= \{(t_{i_1},\ldots,t_{i_k}) \mid (t_1,\ldots,t_n) \in [\![Q]\!]_{\mathcal{D}},\ 1 \le i_1,\ldots,i_k \le n\}\\
[\![\sigma_\theta(Q)]\!]_{\mathcal{D}} &:= \{\mu \mid \mu \in [\![Q]\!]_{\mathcal{D}},\ \mu \models \theta\}\\
[\![Q \times Q']\!]_{\mathcal{D}} &:= \{(\bar{t},\bar{t}') \mid \bar{t} \in [\![Q]\!]_{\mathcal{D}},\ \bar{t}' \in [\![Q']\!]_{\mathcal{D}}\}\\
[\![Q \circ Q']\!]_{\mathcal{D}} &:= [\![Q]\!]_{\mathcal{D}} \circ [\![Q']\!]_{\mathcal{D}}, \quad \text{for } \circ \in \{\cup, \setminus\}
\end{aligned}
$$

PGQ$^{\text{RW}}$ semantics:

$$
\begin{aligned}
[\![c]\!]_{\mathcal{D}} &:= c \quad \text{where } c \in \text{adom}(\mathcal{D})\\
[\![\psi_\Omega(Q_1,\ldots,Q_6)]\!]_{\mathcal{D}} &:= [\![\psi_\Omega]\!]_G \quad \text{where } G := \text{pgView}([\![Q_1]\!]_{\mathcal{D}},\ldots,[\![Q_6]\!]_{\mathcal{D}})
\end{aligned}
$$

PGQ$^{\text{EXT}}$ semantics:

$$
[\![\psi^{\text{ext}}_\Omega(Q_1,\ldots,Q_6)]\!]_{\mathcal{D}} := [\![\psi_\Omega]\!]_G \quad \text{where } G := \text{pgView}^{\text{ext}}([\![Q_1]\!]_{\mathcal{D}},\ldots,[\![Q_6]\!]_{\mathcal{D}})
$$

Fig. 4. Semantics of core PGQ queries.

The semantics of $\psi_\Omega(R_1,\ldots,R_6)$ on a database $\mathcal{D}$ are

$$
[\![\psi_\Omega(R_1,\ldots,R_6)]\!]_{\mathcal{D}} := [\![\psi_\Omega]\!]_G \quad \text{where} \quad G := \text{pgView}\left([\![R_1]\!]_{\mathcal{D}},\ldots,[\![R_6]\!]_{\mathcal{D}}\right).
$$

That is, to evaluate the query $\psi_\Omega(\bar{R})$, we first evaluate each of the six relational subqueries R_1 through R_6 on the current database instance $\mathcal{D}$. These results define the graph G via the pgView operator. Then we evaluate the pattern ψ_Ω over G to produce the result.

4 Read-Write PGQ

In the read-only fragment, pattern matching is restricted to relations that are explicitly stored in the database. In contrast, the read-write fragment extends this by allowing pattern matching to be applied to property graph views constructed dynamically from queries. These queries can be defined recursively, enabling the application of pattern matching not only on the original relations, but also on the results of previous pattern matchings, relational algebra operations, or combinations

thereof. This recursive composition supports a powerful form of query nesting and reuse, central to the expressive power of the read-write fragment.

The syntax and semantics additions for *read-write PGQ queries* can be found in PGQ$^{\text{RW}}$ sections in Figures 3 and 4, respectively.

In particular, the semantics of a query $\psi_\Omega(Q_1, \ldots, Q_6)$ on a database $\mathcal{D}$ is

$$[\![\psi_\Omega(Q_1, \ldots, Q_6)]\!]_{\mathcal{D}} := [\![\psi_\Omega]\!]_G \quad \text{where} \quad G := \mathsf{pgView}\left([\![Q_1]\!]_{\mathcal{D}}, \ldots, [\![Q_6]\!]_{\mathcal{D}}\right).$$

That is, we first evaluate each of the six relational subqueries Q_1 through Q_6 on the current database instance $\mathcal{D}$. These results define the graph G via the pgView operator, and the pattern ψ_Ω is then evaluated over G.

This generalizes the simpler case $\psi_\Omega(\bar{R})$ of PGQ$^{\text{RO}}$, where each R_i is a base relation rather than a query.

We now turn to compare the expressive power of the read-only and read-write fragments.

4.1 Expressiveness of PGQ$^{\text{RW}}$ Vs. PGQ$^{\text{RO}}$

Core PGQ [12] is effectively relational algebra over pattern matching outputs. Since relational algebra is equivalent to relational calculus, it follows that FO $\subseteq$ PGQ$^{\text{RO}}$. Quite expectedly, read-write PGQ queries add expressiveness to the read-only fragment.

Theorem 4.1. *PGQ^{RW} is strictly more expressive than PGQ^{RO}.*

The proof relies on the following separating example. Consider a graph whose vertices are partitioned into `RedNodes` and `BlueNodes`, and whose `Edges` always connect nodes of opposite colors. Detecting a path of *arbitrary* length whose colors alternate red-blue at every step separates the two fragments. In this example, any PGQ$^{\text{RO}}$ query can be rewritten into one that does not apply pattern matching, and therefore in relational algebra. Using Gaifman locality theorem [11], this query can inspect paths only up to a fixed radius determined by its size, so it cannot recognize all unbounded alternating paths. In contrast, PGQ$^{\text{RW}}$ can first materialize a view whose node set is $\texttt{RedNodes} \cup \texttt{BlueNodes}$ and whose edge set is `Edges`, and then apply the reachability pattern

$$(((x) \rightarrow (y) \rightarrow (z)\langle \texttt{RedNodes}(x) \wedge \texttt{BlueNodes}(y) \wedge \texttt{RedNodes}(z)\rangle)^*)_{\emptyset}.$$

This query returns true exactly when an alternating-color path with at least two edges exists, which concludes the proof.

Although PGQ$^{\text{RW}}$ is more expressive than its read-only counterpart, its Boolean queries still do not capture all the problems in NL. This complexity class, NL, is a widely accepted benchmark for graph query languages because it balances expressive power and efficiency. It includes many path-based queries and corresponds to Datalog's capabilities on CRPQs [6], as well as SQL's **`WITH RECURSIVE`**, which supports linear recursion.

Theorem 4.2. *NL is strictly more expressive than Boolean PGQ^{RW}.*

This highlights an expressiveness gap between PGQ$^{\text{RW}}$ and the full range of tractable graph queries. Bridging this gap motivates the extended fragment PGQ$^{\text{EXT}}$ studied next.

5 Extensions of Read-Write PGQ

In the *extended read-write fragment* PGQ$^{\text{EXT}}$, whose concrete syntax and formal semantics are set out in Figures 3 and 4, we lift the restriction that node and edge identifiers are single values: they may now be n-ary tuples for any fixed $n \geq 1$. With this single generalization, each clause of Definition 3.2 carries over (1)-(4) with the relation arities scaled accordingly.

Definition 5.1. An *n-ary property graph view* is a relational database $\mathcal{D}$ over a schema consisting of six relation symbols-n-ary R_1, R_2, $2n$-ary R_3, R_4, an $(n+1)$-ary R_5, and an $(n+2)$-ary R_6-for which conditions (1)-(4) from Definition 3.2 hold. That is,

(1) $R_1^{\mathcal{D}}$ and $R_2^{\mathcal{D}}$ are disjoint relations (representing node and edge identifiers, respectively).
(2) $R_3^{\mathcal{D}}, R_4^{\mathcal{D}}$ encode functions $R_2^{\mathcal{D}} \to R_1^{\mathcal{D}}$ (representing the source and target of each edge, respectively).
(3) $R_5^{\mathcal{D}} \subseteq (R_1^{\mathcal{D}} \cup R_2^{\mathcal{D}}) \times C$ (representing the labeling of nodes and edges).
(4) $R_6^{\mathcal{D}}$ encodes a partial function $(R_1^{\mathcal{D}} \cup R_2^{\mathcal{D}}) \times C \rightharpoonup C$ (representing properties).

Indeed, this is a generalization of Definition 3.2, since for $n = 1$ the two definitions coincide.

Remark 5.1. In our definition of n-ary property graphs, node and edge identifiers share the same arity to simplify the model: a single pair of relations R_5, R_6 suffices for labels and properties. This choice reduces complications without loss of generality. Allowing different arities for nodes and edges requires duplicating these relations, but all definitions and results extend naturally to that case.

Definition 5.2. For every integer $n \geq 1$, we define $\mathsf{pgView}^{=n}$ as the partial function that, given a sequence of relations $(\mathbf{R}_1, \ldots, \mathbf{R}_6)$ that satisfy the n-ary property graph view (Definition 5.1), returns the corresponding property graph $\langle N, E, \mathsf{src}, \mathsf{tgt}, \mathsf{lab}, \mathsf{prop}\rangle$, denoted by $\mathsf{pgView}^{=n}(\mathbf{R}_1, \ldots, \mathbf{R}_6)$ where

$$N := \mathbf{R}_1, \quad E := \mathbf{R}_2, \quad \mathsf{src} := \mathbf{R}_3, \quad \mathsf{tgt} := \mathbf{R}_4, \quad \mathsf{lab} := \mathbf{R}_5, \quad \mathsf{prop} := \mathbf{R}_6.$$

Definition 5.3. For every integer $n \geq 1$, we set pgView^{n} to be the union of $\mathsf{pgView}^{=i}$ up to $i = n$, that is,

$$\mathsf{pgView}^{n} := \bigcup_{i=1}^{n} \mathsf{pgView}^{=i},$$

and $\mathsf{pgView}^{\mathrm{ext}}$ as the union of all i's:

$$\mathsf{pgView}^{\mathrm{ext}} := \bigcup_{i \geq 1} \mathsf{pgView}^{=i}.$$

That is, $\mathsf{pgView}^{\mathrm{ext}}$ is the partial function whose domain is the union of the domains of all pgView^{n}. Concretely, for every sequence of relations $(\mathbf{R}_1, \ldots, \mathbf{R}_6)$ that satisfies the k-ary property graph view conditions for some $k \geq 1$, we set

$$\mathsf{pgView}^{\mathrm{ext}}(\mathbf{R}_1, \ldots, \mathbf{R}_6) := \mathsf{pgView}^{k}(\mathbf{R}_1, \ldots, \mathbf{R}_6).$$

Notice that $\mathsf{pgView}^{1} = \mathsf{pgView}$, hence pgView^{k} extends pgView. Note also that $\mathsf{pgView}^{\mathrm{ext}}$ is well defined.

We define the extended read-write PGQ, namely $\mathrm{PGQ}^{\mathrm{EXT}}$ by adding $\psi_\Omega^{\mathrm{ext}}(\bar{Q})$ to the syntax grammar. A query $Q \in \mathrm{PGQ}^{\mathrm{EXT}}$ is defined by the additional syntax in the $\mathrm{PGQ}^{\mathrm{EXT}}$ section in Figure 3:

$$Q := \psi_\Omega^{\mathrm{ext}}(Q_1, \ldots, Q_6).$$

The semantics is defined using $\mathsf{pgView}^{\mathrm{ext}}$ as follows:

$$[\![\psi_\Omega^{\mathrm{ext}}(Q_1, \ldots, Q_6)]\!]_{\mathcal{D}} := [\![\psi_\Omega]\!]_G, \quad \text{where } G = \mathsf{pgView}^{\mathrm{ext}}([\![Q_1]\!]_{\mathcal{D}}, \ldots, [\![Q_6]\!]_{\mathcal{D}}),$$

and can be found in $\mathrm{PGQ}^{\mathrm{EXT}}$ section in Figure 4. $\mathrm{PGQ}^{\mathrm{EXT}}$ preserves the two-phase evaluation procedure of $\mathrm{PGQ}^{\mathrm{RW}}$ but replaces the operator pgView with its composite-identifier variant $\mathsf{pgView}^{\mathrm{ext}}$, and substitutes the core pattern ψ_Ω with the extended pattern $\psi_\Omega^{\mathrm{ext}}$, whose semantics supports identifiers of arity k. In particular, in Figure 2, each output triple (s, t, μ) has s and t as k-tuples (rather than arity-1), valuations μ map variables to k-tuples, and all equalities in the n-fold repetition are over k-tuples.

Expressiveness. Since every PGQ^{RW} query is also an PGQ^{EXT} query, we have the immediate containment $\text{PGQ}^{\text{RW}} \subseteq \text{PGQ}^{\text{EXT}}$. The strictness of this containment is not evident from the syntax alone, but will follow from the capture result proved in the next section and from Theorem 4.2.

Example 5.1 (Extended Graph View with Composite Identifiers). Building on Examples 1.1 and 2.1, assume now that accounts in our database are no longer identified by a single IBAN column. Instead, an account is uniquely identified by the triple (`bank`, `branch`, `acct`). Thus, the relational schema is adapted, replacing the IBAN field by three separate columns:

- `Account(bank, branch, acct)` identifies accounts by triples.
- `Transfer(t_id, bankSrc, branchSrc, acctSrc, bankTgt, branchTgt, acctTgt, ts, amount)` stores transfers between accounts identified by triples.

To capture this structure, we define the following property graph view with composite n-ary identifiers for nodes and edges:

R_1 is the table `Account(bank,branch,acct)`; R_2 is the projection of `Transfer` onto `t_id`; R_3 links every edge in R_2 to its source account `(bankSrc, branchSrc,acctSrc)`; R_4 links every edge in R_2 to its target account `(bankTgt, branchTgt, acctTgt)`; R_5 assigns the label `Transfer` to each edge in R_2; finally, R_6 is empty in this example, i.e. $R_6 = \emptyset$.

Using composite identifiers significantly simplifies pattern matching queries. For instance, one can succinctly write:

$$\left((x) \xrightarrow{t}{}^{1,\infty} (y)\langle\theta\rangle\right)_{x.\mathsf{bank},x.\mathsf{branch},y.\mathsf{bank},y.\mathsf{branch}} \qquad \text{where} \quad \theta := \mathtt{Transfer}(t)$$

The output of this query includes the identifiers of banks and branches of source and target accounts, which we can subsequently filter on without additional joins. By contrast, in Example 2.1 the output would have contained only the long IBAN strings for the endpoints, preventing such post filtering.

Theorem 5.2. *PGQ^{EXT} is strictly more expressive than PGQ^{RW}.*

The PGQ^{EXT} fragment closes a gap in the expressiveness identified in [12]. We discuss it in the following example:

Example 5.3 (Increasing Values on Edges). Consider again the graph view defined in Example 5.1. Suppose we want to find all pairs of accounts connected by paths of transfers such that the transferred amounts strictly increase along the path edges. This was shown to be inexpressible by output patterns in [12]. We construct a new property graph in the PGQ^{EXT} fragment as follows: For each account node `(bank,branch,acct)`, we create multiple copies, for each incoming amount `l` transferred to it. So now the identifier is `(bank,branch,acct,l)`.

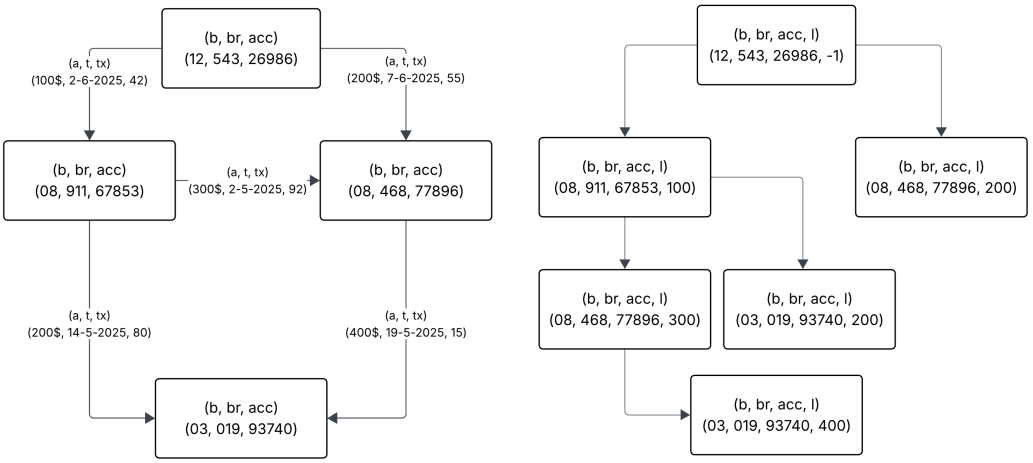
Edges in the new graph connect node copies in a strictly increasing manner: an edge connects node `(bank,branch,acct,l)` to node `(bank,branch,acct,j)` only if `j>l`. See Figure 5 for illustration.

Now, querying such paths is straightforward using standard path pattern matching.

We do not need to add a filter enforcing increasing amounts, the composite identifiers already encode this information at every node copy, so any path produced by the query is guaranteed to follow strictly increasing transfer amounts.

This demonstrates the expressive capability of the PGQ^{EXT} fragment: by enriching the natural identifier we can materialize multiple copies of the same account, each copy representing the node in a different state, determined by the incoming transfer amount.

We now turn to a more fine-grained analysis of the language PGQ^{EXT}.



(a) Input property graph G (b) Constructed property graph G'

Fig. 5. Illustration of the view-construction step in Example 5.3.

6 Expressiveness Results

To characterize the expressive power of PGQ$^{\text{EXT}}$, we relate it to FO[TC]. This connection establishes PGQ$^{\text{EXT}}$ as a robust graph query language and positions it precisely within the classical complexity class NL.

6.1 Equivalence of PGQ$^{\text{EXT}}$ and FO[TC]

Our goal now is to establish a precise correspondence between the expressive power of the extended read-write fragment PGQ$^{\text{EXT}}$ and first-order logic with transitive closure, FO[TC]. We proceed by formally recalling the semantics of FO[TC], and then prove the equivalence.

FO[TC] overview. First-order (FO) formulas over a relational schema $\mathcal{S}$ are constructed from atomic formulas of the form $R(x_1, \ldots, x_n)$ where $R \in \mathcal{S}$ has arity n, and from equalities $x = y$. Composite formulas are then built using the standard Boolean connectives $\neg$, $\wedge$, $\vee$, along with the quantifiers $\exists$ and $\forall$.

Given a database instance $\mathcal{D}$ over $\mathcal{S}$, and a tuple of values $\bar{a}$ corresponding to a tuple of variables $\bar{x}$, we write $\mathcal{D} \models \varphi[\bar{a}/\bar{x}]$ to indicate that the formula $\varphi(\bar{x})$ is satisfied in $\mathcal{D}$ when each free variable in $\bar{x}$ is replaced by the corresponding value in $\bar{a}$.

A tuple $\bar{a}$ satisfies $\varphi(\bar{x})$ in a database instance $\mathcal{D}$ (written $\mathcal{D} \models \varphi[\bar{a}/\bar{x}]$) when replacing the free variables $\bar{x}$ by the concrete values $\bar{a}$ makes the sentence true.

FO[TC] extends FO with the transitive-closure operator

$$\mathrm{TC}_{(\bar{u},\bar{v})}\big[\psi(\bar{u},\bar{v},\bar{p})\big](\bar{x},\bar{y}), \qquad |\bar{u}| = |\bar{v}| = |\bar{x}| = |\bar{y}|.$$

The formula holds in a database $\mathcal{D}$ under assignment $[\bar{a},\bar{b},\bar{c}/\bar{u},\bar{v},\bar{p}]$ if there exists a finite sequence of tuples $\bar{a} = \bar{a}_0, \bar{a}_1, \ldots, \bar{a}_n = \bar{b}$ such that for each $i < n$, $\mathcal{D} \models \psi[\bar{a}_i, \bar{a}_{i+1}, \bar{c}/\bar{u},\bar{v},\bar{p}]$. That is, ψ relates each consecutive pair in the sequence, using the same parameter tuple $\bar{c}$, and the overall formula expresses reachability under the binary relation defined by ψ.

For any FO[TC] formula $\varphi(\bar{x})$ we write

$$[\![\varphi(\bar{x})]\!]_{\mathcal{D}} \;:=\; \{\; \bar{a} \mid \mathcal{D} \models \varphi[\bar{a}/\bar{x}] \;\},$$

that is, the *result relation* consisting of all tuples that make φ true in $\mathcal{D}$.

With these definitions, we can now show the equivalence $\text{PGQ}^{\text{EXT}} = \text{FO[TC]}$ by proving that each language subsumes the other.

Theorem 6.1 ($\text{PGQ}^{\text{EXT}} \subseteq \text{FO[TC]}$). *For every schema $\mathcal{S}$ and for every query $Q \in PGQ^{EXT}$ over $\mathcal{S}$ with* $\text{arity}(Q) = n$, *there exists an* FO[TC] *formula $\varphi_Q(x_1, \ldots, x_n)$ over the same schema $\mathcal{S}$ such that*

$$[\![Q]\!]_{\mathcal{D}} = [\![\varphi_Q(x_1, \ldots, x_n)]\!]_{\mathcal{D}}$$

for every relational database $\mathcal{D}$ over $\mathcal{S}$.

In the proof, we decompose the translation from PGQ^{EXT} to FO[TC] into two steps. (i) Algebraic core: union, difference, Cartesian product, projection, and selection are mapped naturally to first-order connectives, laying the base of the induction. (ii) Pattern matching: we take a path pattern ψ and produce an FO[TC] formula $\varphi_\psi(\bar{y}, \bar{x}_{\text{src}}, \bar{x}_{\text{tgt}})$ that incorporates two variables $\bar{x}_{\text{src}}, \bar{x}_{\text{tgt}}$ for the source and target, respectively, of the matched paths, and whose other free variables $\bar{y}$ explicitly track every binding in the semantics of ψ. This translation coincides with the one in [8] up to minor notational differences.

The induction stitches these two parts (i) and (ii) together to prove the semantics equivalence. As an illustration of step (ii), consider the Kleene-star pattern ψ^*. Assume we have the translation $\tau(\psi)(\bar{x}, \bar{u}, \bar{v})$ of ψ with free variables are $\bar{x}, \bar{u}, \bar{v}$. Due to the semantics of path patterns (Figure 2), the translation $\tau(\psi^*)$ has free variables $\bar{x}_{\text{src}}, \bar{x}_{\text{tgt}}$ and is defined recursively as follows:

$$\tau(\psi^*)(\bar{x}_{\text{src}}, \bar{x}_{\text{tgt}}) \;=\; \exists \bar{x} : \Big(\text{TC}_{\bar{u},\bar{v}}\ \tau(\psi)(\bar{x}, \bar{u}, \bar{v})\Big)(\bar{x}_{\text{src}}, \bar{x}_{\text{tgt}}),$$

This concrete instance demonstrates how a recursive graph pattern is captured in FO[TC] by wrapping the translation of the base pattern ψ inside an appropriate transitive closure.

The other direction also holds as the next theorem shows.

Theorem 6.2 ($\text{FO[TC]} \subseteq \text{PGQ}^{\text{EXT}}$). *For every schema $\mathcal{S}$, and every* FO[TC] *formula $\varphi(x_1, \ldots, x_n)$ over $\mathcal{S}$, there exists a query $Q_\varphi \in PGQ^{EXT}$ over $\mathcal{S}$ with* $\text{arity}(Q_\varphi) = n$ *such that*

$$[\![\varphi(x_1, \ldots, x_n)]\!]_{\mathcal{D}} \;=\; [\![Q_\varphi]\!]_{\mathcal{D}}$$

for every relational database $\mathcal{D}$ over $\mathcal{S}$.

The proof is based on a full translation T from FO[TC] to PGQ^{EXT}. First-order connectives and quantifiers are translated inductively in a straightforward manner. The key step is the translation of transitive closure subformulas:

$$\psi \;=\; \text{TC}_{(\bar{u},\bar{v})}\big[\varphi(\bar{u}, \bar{v}, \bar{p})\big](\bar{x}, \bar{y}), \qquad |\bar{x}| = |\bar{y}| = |\bar{u}| = |\bar{v}|.$$

Note that the parameters $\bar{p}$ remain fixed across all iterations of the transitive closure. By the inductive hypothesis, we obtain the translation $T(\varphi)$ of φ, and define $C := \pi_{\bar{p}}(T(\varphi))$, the projection onto the parameter values.

Then, for each concrete parameter tuple $\bar{c} \in C$, we isolate the corresponding $\bar{u}, \bar{v}$ pairs and use the extended graph constructor from Section 5, namely $\text{pgView}^{\text{ext}}$, to define the new graph $G_{\bar{c}}$.

We then apply the reachability pattern: $\psi_{reach} := ((\bar{x}) \rightarrow^* (\bar{y}))_{\bar{x},\bar{y}}$ to $G_{\bar{c}}$. This returns all $(\bar{x}, \bar{y})$ pairs connected by a path in $G_{\bar{c}}$. We define the overall query:

$$Q_\psi := \bigcup_{\bar{c} \in C} \Big(\psi_{reach}(G_{\bar{c}}) \times \{\bar{c}\}\Big), \qquad \text{fv}\,(Q_\psi) = (\bar{x}, \bar{y}).$$

Note that this union is realized by an ordinary join $\psi_{reach}(G_{\bar{c}}) \bowtie \sigma_{\bar{p}=\bar{c}}(C)$. By construction, $[\![Q_\psi]\!]_{\mathcal{D}} = [\![\psi]\!]_{\mathcal{D}}$ for every database $\mathcal{D}$, so Q_ψ can be treated as an atomic subquery by outer Boolean connectives and quantifiers.

Combining Theorems 6.1 and 6.2, we obtain

Corollary 6.3 (Expressive equivalence). *In terms of expressiveness,* $PGQ^{EXT} = \mathrm{FO[TC]}$.

This leads to the following straightforward observation.

Corollary 6.4 (NL capture). *The evaluation problem for* PGQ^{EXT} *is NL complete.*

We now dive deeper into the language $\mathrm{PGQ^{EXT}}$ to better understand what it is that enables it to capture every query definable in FO[TC].

6.2 Inside $\mathrm{PGQ^{EXT}}$

What makes $\mathrm{PGQ^{EXT}}$ more expressive than $\mathrm{PGQ^{RW}}$? To answer this, we adopt a more fine-grained analysis of the language's capabilities. Recall from Definition 5.1 the function

$$\mathsf{pgView}^{\mathrm{ext}} = \bigcup_{n \geq 1} \mathsf{pgView}^{=n},$$

which allows node and edge identifiers of *any* arity. Higher arities enable richer view definitions, which in turn unlock additional expressive power not available in the unary-identifier setting of $\mathrm{PGQ^{RW}}$.

For every fixed integer $n \geq 1$, we define PGQ^n as the language obtained from $\mathrm{PGQ^{EXT}}$ by replacing $\psi^{\mathrm{ext}}_{\Omega}(\bar{Q})$ with $\psi^{(n)}_{\Omega}(\bar{Q})$, and set the semantics to be

$$\left[\!\left[\psi^{(n)}_{\Omega}(\bar{Q})\right]\!\right]_{\mathcal{D}} := [\![\psi_{\Omega}]\!]_G, \quad \text{where } G = \mathsf{pgView}^n\left([\![Q_1]\!]_{\mathcal{D}}, \ldots, [\![Q_6]\!]_{\mathcal{D}}\right)$$

and $\bar{Q} := (Q_1, Q_2, Q_3, Q_4, Q_5, Q_6)$.

On the FO[TC] side, we write $\mathrm{FO[TC^n]}$ to denote the fragment of first-order logic with transitive closure in which all transitive closure operators

$$\mathrm{TC}_{\bar{u},\bar{v}}[\psi(\bar{u}, \bar{v}, \bar{p})](\bar{x}, \bar{y})$$

use tuples $\bar{u}, \bar{v}, \bar{x}, \bar{y}$ of fixed arity n.

With this notation, we can show that PGQ^n translates to $\mathrm{FO[TC^n]}$.

Theorem 6.5 ($\mathrm{PGQ}^n \subseteq \mathrm{FO[TC^n]}$). *Let* $n \geq 1$. *For every schema* $\mathcal{S}$ *and every query* $Q \in PGQ^n$ *over* $\mathcal{S}$ *with* $\mathsf{arity}(Q) = k$, *there exists an* $\mathrm{FO[TC^n]}$ *formula* $\varphi_Q(x_1, \ldots, x_k)$ *such that*

$$[\![Q]\!]_{\mathcal{D}} = \left[\!\left[\varphi_Q(x_1, \ldots, x_k)\right]\!\right]_{\mathcal{D}}$$

for every relational database $\mathcal{D}$ *over* $\mathcal{S}$.

The converse translation also exists.

Theorem 6.6 ($\mathrm{FO[TC^n]} \subseteq \mathrm{PGQ}^n$). *Let* $n \geq 1$. *For every schema* $\mathcal{S}$ *and every* $\mathrm{FO[TC^n]}$ *formula* $\varphi(x_1, \ldots, x_k)$ *over* $\mathcal{S}$, *there exists a query* $Q_\varphi \in PGQ^n$ *over* $\mathcal{S}$ *with* $\mathsf{arity}(Q_\varphi) = k$ *such that*

$$[\![\varphi(x_1, \ldots, x_k)]\!]_{\mathcal{D}} = \left[\!\left[Q_\varphi\right]\!\right]_{\mathcal{D}}$$

for every relational database $\mathcal{D}$ *over* $\mathcal{S}$.

Both Theorems 6.5 and 6.6 are obtained by inductive syntax-directed translations.

These are refined versions of the unary translations already given in Theorems 6.1 and 6.2 to the n-ary case. For $\mathrm{PGQ}^n \subseteq \mathrm{FO[TC^n]}$, we generalize the translation from Theorem 6.1 to ensure the resulting formula is in $\mathrm{TC^n}$. For $\mathrm{FO[TC^n]} \subseteq \mathrm{PGQ}^n$, we generalize the translation from Theorem 6.2 to ensure the resulting query is in PGQ^n. This gives desired equivalence $\mathrm{PGQ}^n = \mathrm{FO[TC^n]}$.

Combining Theorems 6.5 and 6.6, we obtain the following.

Corollary 6.7 (Expressive equivalence for fixed arity). *In terms of expressiveness,*

$$PGQ^n = \mathrm{FO}[\mathrm{TC}^n] \qquad \text{for every } n \geq 1.$$

We can derive the following expressiveness chain on fragments within $\mathrm{PGQ}^{\mathrm{EXT}}$.

Theorem 6.8 (Expressiveness chain).

$$PGQ^{RW} = PGQ^1 = \mathrm{FO}[\mathrm{TC}^1] \subsetneq \mathrm{FO}[\mathrm{TC}^2] = \mathrm{FO}[\mathrm{TC}^n] = PGQ^{EXT}.$$

Proof. The chain begins with the observation that the read-write fragment is, by definition, the same as the arity-1 extension; hence $\mathrm{PGQ}^{\mathrm{RW}} = \mathrm{PGQ}^1$. Corollary 6.7 then equates this fragment with unary transitive-closure logic, giving $\mathrm{PGQ}^1 = \mathrm{FO}[\mathrm{TC}^1]$. By Immerman's collapse theorem for ordered structures, unary TC is strictly weaker than binary TC, and every higher arity collapses to the binary case, formally, $\mathrm{FO}[\mathrm{TC}^1] \subsetneq \mathrm{FO}[\mathrm{TC}^2] = \mathrm{FO}[\mathrm{TC}^n]$ for all $n \geq 2$ [15]. The same corollary yields $\mathrm{FO}[\mathrm{TC}^2] = \mathrm{PGQ}^2$, so every construct definable in any higher-arity fragment PGQ^n can already be written in PGQ^2. Because $\mathrm{PGQ}^{\mathrm{EXT}}$ allows identifiers of arity n, we have $\mathrm{PGQ}^n = \mathrm{PGQ}^{\mathrm{EXT}}$. Combining these equalities gives the desired chain. □

7 Comparison with the SQL/PGQ Standard

We now clarify how our theoretical framework relates to the SQL/PGQ standard.

Property Graph Views. The ISO draft [16] distinguishes a *pure property graph (PPG)* from its underlying *tabular property graph (TPG)*. In our framework, PPG coincides with the property graph of Definition 2.1, whereas TPG with property graph view of Definition 3.2. We represent property graph views by the canonical relations $(R_1, \ldots, R_6)$ that form the inputs to the pgView family (Definitions 3.2-5.3). This choice differs from the Standard's *vertex-table/edge-table* notation:

(1) *Normalization:* The six-relation encoding we use is highly normalized: each elementary fact, that is, vertex or edge identifier, source, target, label, or property, occurs only once. By comparison, a single vertex table can repeat the same label and property values across many rows, and an edge table can replicate source or target keys. Strictly adopting the vertex-edge-table layout of the ISO draft may affect our results.
(2) *Identifiers:* The standard requires each vertex and edge row to carry a primary key. This is reflected by R_1 (vertex identifiers) and R_2 (edge identifiers) in our model, ensuring that query results can always return the graph elements, that is, its node and edge IDs.
(3) *Arity of Identifiers:* Our core read-write fragment $\mathrm{PGQ}^{\mathrm{RW}}$ assumes unary identifiers, matching the simplest TPGs whose keys are single columns. Our extended fragment $\mathrm{PGQ}^{\mathrm{EXT}}$ generalizes this to n-ary identifiers, covering n-ary keys allowed by the standard but keeping them outside the core language. This separation reflects the usual treatment of relational algebra: formal models of core SQL deal only with single, atomic attributes, whereas the full SQL standard goes further and also allows composite (multi-attribute) types.
(4) *Labels:* Unlike the standard, our model does not require every edge to have a label. Under our definitions, an edge may be unlabeled, that is, it can appear in the graph without any associated label entry in R_5. To align fully with the ISO draft, one may simply insert a distinguished default label for such edges.

Updates. We omit update operations on TPGs to keep a clean theoretical core. This causes no loss of generality: any change can be simulated by rebuilding the six base relations and reapplying pgView (or its extensions), so all results in this paper still hold.

8 Conclusion and Future Work

We have characterized the expressive power of SQL/PGQ, showing how its graph view mechanism fundamentally shapes the language's capabilities. Our analysis identifies three fragments, PGQ^{RO}, PGQ^{RW}, and PGQ^{EXT}, and places them in a strict expressiveness hierarchy culminating in NL, captured precisely by the extended fragment PGQ^{EXT}. While PGQ^{EXT} matches this foundational benchmark, it is not yet clear how naturally its expressiveness translates into usable query syntax. For example, the "increasing values on edges" query provably inexpressible in the pattern matching layer [12], can be expressed in PGQ^{EXT} using composite identifiers and dynamic graph construction as described in Example 5.3. However, the required encoding is nontrivial. To express that query, we simulate value tracking by creating node copies, each annotated with the amount of the incoming edge. Edges connect only copies where the target amount is greater than the amount of the source. This is expressible in PGQ^{EXT} and demonstrates its ability to capture complex value-based recursion but also highlights that the encoding is intricate and unlikely to be intuitive for users. Whether such queries can be formulated more naturally within PGQ^{EXT}, or whether new syntax is needed to express them cleanly, remains an open question. Finally, our formalization opens the door to compositional graph-query languages: pgView constructs full property graphs that can be queried or outputted. In addition to compositionality, our model offers a natural foundation for unified languages that query both relations and graph databases, and can move naturally and seamlessly from one model to the other.

Acknowledgments

The authors were supported by ISF grant 2355/24.

References

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1421–1432. doi:10.1145/3183713.3190654

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[3] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.* 37, 4 (2012), 31:1–31:46.

[4] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying regular graph patterns. *Journal of the ACM* 61, 1 (2014), 8:1–8:54.

[5] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. 2012. Relative Expressiveness of Nested Regular Expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2012) (CEUR Workshop Proceedings, Vol. 866)*, Juliana Freire and Dan Suciu (Eds.). CEUR-WS.org, Aachen, Germany, 180–195. https://ceur-ws.org/Vol-866/paper13.pdf

[6] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman (Eds.). Morgan Kaufmann, San Francisco, CA, USA, 176–185. Breckenridge, CO, USA, April 11–15, 2000.

[7] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, New York, NY, USA, 323–330. doi:10.1145/38713.38749

[8] Diego Figueira, Anthony W. Lin, and Liat Peterfreund. 2024. Relational Perspective on Graph Query Languages. *CoRR* abs/2407.06766 (2024). arXiv:2407.06766 doi:10.48550/arXiv.2407.06766

[9] 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 *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Floris Geerts, Hung Q. Ngo, and Stavros Sintos (Eds.). ACM, Seattle, WA, USA, 241–250. doi:10.1145/3584372.3588662

[10] 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. In *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece (LIPIcs, Vol. 255)*, Floris Geerts and Brecht Vandevoort (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:22. doi:10.4230/LIPICS.ICDT.2023.1

[11] Haim Gaifman. 1982. On Local and Non-Local Properties. In *Proceedings of the Herbrand Symposium*, J. Stern (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 107. Elsevier, Amsterdam, 105–135. doi:10.1016/S0049-237X(08)71879-2

[12] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proceedings of the VLDB Endowment (PVLDB)* 18, 6 (2025), 1798–1810. doi:10.14778/3725688.3725707 Licensed under CC BY-NC-ND 4.0.

[13] GQL Standards Committee. 2024. GQL Standards Website. https://www.gqlstandards.org/ Accessed: November 2024.

[14] Alastair Green, Paolo Guagliardo, and Leonid Libkin. 2021. *Property graphs and paths in GQL: Mathematical definitions.* Technical Reports TR-2021-01. Linked Data Benchmark Council (LDBC). doi:10.54285/ldbc.TZJP7279

[15] Neil Immerman. 1999. *Descriptive Complexity.* Springer, New York.

[16] ISO/IEC JTC 1/SC 32/WG 3. 2023. ISO/IEC 9075-16: SQL/PGQ - Property Graph Queries. https://www.iso.org/standard/79473.html. Working Draft, ISO/IEC JTC 1/SC 32/WG 3.

[17] 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*, Floris Geerts and Benny Kimelfeld (Eds.). ACM, New York, NY, USA, 9–26. doi:10.1145/3722234.3725822

[18] Batul J Mirza, Benjamin J Keller, and Naren Ramakrishnan. 2003. Studying recommendation algorithms by graph analysis. *Journal of intelligent information systems* 20 (2003), 131–160.

[19] openCypher. 2017. Cypher Query Language Reference, Version 9. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf

[20] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2015. Regular Queries on Graph Databases. In *18th International Conference on Database Theory (ICDT 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 31)*, Marcelo

Arenas and Martín Ugarte (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Brussels, Belgium, 177–194. doi:10.4230/LIPIcs.ICDT.2015.177

[21] Sakshi Srivastava and Anil Kumar Singh. 2023. Fraud detection in the distributed graph database. *Cluster Computing* 26, 1 (2023), 515–537.

[22] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Rec.* 53, 2 (July 2024), 21–37. doi:10.1145/3685980.3685984

[23] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. doi:10.1145/2960414.2960421

Received June 2025; accepted August 2025