


Database Theory in Action: From Inexpressibility to Efficiency in GQL’s Order-Constrained Paths

Hadar Rotschild ✉ 

School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel

Liat Peterfreund ✉ 

School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel

Abstract

Pattern matching of core GQL, the new ISO standard for querying property graphs, cannot check whether edge values are increasing along a path, as established in recent work. We present a constructive translation that overcomes this limitation by compiling the increasing-edges condition into the input graph. Remarkably, the benefit of this construction goes beyond restoring expressiveness. In our proof-of-concept implementation in Neo4j’s Cypher, where such path constraints are expressible but costly, our compiled version runs faster and avoids timeouts. This illustrates how a theoretically motivated translation can not only close an expressiveness gap but also bring practical performance gains.

2012 ACM Subject Classification Theory of computation → Database theory; Information systems → Graph-based database models; Information systems → Query languages

Keywords and phrases Property graphs, ISO GQL, Graph Query Languages, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.26

Related Version *Full Version:* <https://arxiv.org/html/2512.23330v1>

Supplementary Material

Software (Source Code): <https://github.com/hadarrot/Order-Constrained-Paths-in-Leveled-Graphs> [8], archived at `swh:1:dir:30effa7157a9e9bbf79b73b574a5189d5b5abef4`

Funding Both authors were supported by ISF grant 2355/24.

1 Introduction

GQL is the new ISO standard for querying property graphs [5]. Property graphs are a rich and flexible data model in which nodes and edges can carry labels (often representing their types) and properties (in a key-value format), which makes them suitable for various domains such as finance, social networks, and knowledge graphs [9, 1, 11].

The GQL standard is lengthy and complex, first formalized in [2, 3], later distilled into its core [4], enabling precise expressiveness analysis, and subsequently positioned on a theory/practice spectrum as articulated in [6]. GQL comprises a pattern-matching layer, used in MATCH to produce bindings of nodes or edges, and a relational layer that post-processes those bindings using relational algebra. One of the key findings in [4] is that queries identifying paths with increasing edge values are not expressible in core GQL. Such queries are common in practice, leading participants in the ISO GQL design to consider possible language extensions to support them [10, 7]. For example, in a financial transaction graph, accounts are nodes with properties such as balance, and transfers are edges with properties such as amount and timestamp. While pattern matching allows to query for chains of transfers where account balances increase along the path, expressing that timestamps increase along a path is impossible. In other words, increasing value conditions can be checked on nodes, but not on edges.



© Hadar Rotschild and Liat Peterfreund;
licensed under Creative Commons License CC-BY 4.0
29th International Conference on Database Theory (ICDT 2026).

Editors: Balder ten Cate and Maurice Funk; Article No. 26; pp. 26:1–26:5

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we present a constructive translation that overcomes the expressiveness gap by compiling an input graph and order condition into a leveled graph which, together with a reachability query, captures the intended semantics. Our contribution is twofold: first, we provide a principled method that makes inexpressible queries formally definable; second, we show that the same approach yields practical benefits. While such queries can be expressed in the full versions of GQL and in Cypher, the formulations are highly convoluted and require reasoning over exponentially many paths, leading to impractical runtimes as demonstrated in the experiments in [4]. In contrast, our technique achieves efficient execution in practice, as confirmed by a proof-of-concept implementation. This demonstrates how a theoretically motivated translation can both extend the expressive power of a standard language and provide concrete performance improvements.

2 Compiling Ordered-Path Constraints into a Leveled Graph

In this section we show how to compile the increasing values along edges condition into the input graph. We start by defining the data model. We assume pairwise disjoint sets *Nodes*, *Edges*, *Labels*, *Properties*, and *Values*, and define a *labeled property graph* (see, e.g., [6]) as a tuple $G = (N, E, \text{src}, \text{tgt}, \lambda, \rho)$ where $N \subseteq \text{Nodes}$ and $E \subseteq \text{Edges}$ are finite, $\text{src}, \text{tgt} : E \rightarrow N$ are total functions, $\lambda : E \rightarrow \text{Labels}$ assigns an edge label to each edge, and $\rho : (N \cup E) \times \text{Properties} \rightarrow \text{Values}$ is a partial function that stores node/edge properties.

Problem Definition. To formulate the problem, we assume each edge is associated with a designated property *val* whose value is in \mathbb{R} . Formally, we assume $\rho(e, \text{val})$ is defined for every $e \in E$ and that $\rho(e, \text{val}) \in \mathbb{R}$. For readability, we denote $u \xrightarrow{j} v$ if there is a directed edge e from u to v with $j := \rho(e, \text{val})$. It was shown in [4] that the following problem is not expressible in pattern matching of core GQL:

Strictly Increasing-Path Existence. For every labeled property graph G , and nodes $s, t \in N$, does there exist a non-empty directed path $s = u_0 \xrightarrow{j_1} u_1 \xrightarrow{j_2} \dots \xrightarrow{j_m} u_m = t$ ($m \geq 1$) whose edge-values are strictly increasing (i.e., $j_1 < j_2 < \dots < j_m$)?

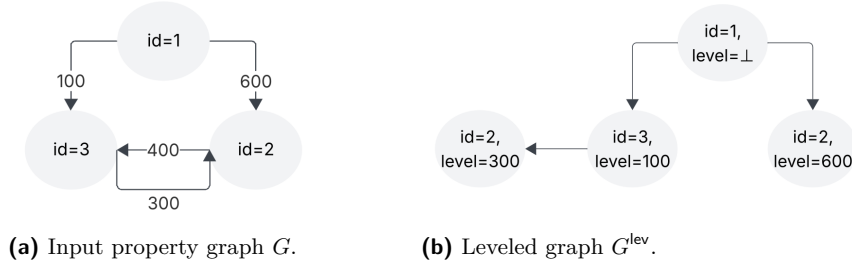
The underlying reason that this query is not expressible is that GQL's concatenation, by design, verifies only node equivalence.

Our Construction

We show that given a graph, we can compile the increasing value condition into it. The idea is to encode, in the node identifier of the compiled graph, the value of an incoming edge, hence reducing the increasing value conditions to ordinary reachability.

► **Definition 1 (Leveled graph).** Let G be a labeled property graph. For each node v of G , let $L(v) = \{j \mid u \xrightarrow{j} v \text{ for some } u\} \cup \{\perp\}$, that is, $L(v)$ is the set of values of the incoming edges to v , together with a special value \perp used for paths that start at v . We assume $\perp < j$ for all values j . The leveled graph G^{lev} of G consists of nodes (v, j) for every edge $u \xrightarrow{j} v$, and, for every $v \in N$, the node (v, \perp) , and edges $(u, \ell) \rightarrow (v, j)$ for every edge $u \xrightarrow{\ell} v$, and $\ell \in L(u)$ with $\ell < j$.

To illustrate the definition, consider the following example.



■ **Figure 1** Levelled-graph example (with amounts). In (a) the increasing path $1 \rightarrow 3 \rightarrow 2$ exists ($100 < 300$), while $1 \rightarrow 2$ with 600 cannot be extended via $2 \rightarrow 3$ with 400. In (b) nodes are copied by their last-seen from node 1 to some node 2 in G^{lev} .

► **Example 2** (Increasing amounts). The graph in Figure 1a describes transfers between accounts where j denotes the amount of the transfer for each edge $u \xrightarrow{j} v$. In G^{lev} in Figure 1b, each account v is replicated as (v, ℓ) for every $\ell \in L(v)$. There is an edge $(u, \ell) \rightarrow (v, j)$ iff there is a transfer $u \xrightarrow{j} v$ in G and there exists $\ell \in L(u)$ for which $\ell < j$. Thus, an increasing-transfer path from s to t in G corresponds to a reachability problem in G^{lev} .

With this intuition, we can move to showing the correctness of our construction.

► **Theorem 3.** Fix a property graph G and nodes $s, t \in N$. There exists a non-empty strictly increasing path $s \rightsquigarrow t$ in G iff there exists $\ell \in L(t)$ such that $(s, \perp) \rightsquigarrow (t, \ell)$ in G^{lev} .

Proof Sketch.

(\Rightarrow) Let $s \rightsquigarrow t$ in G be $u_0 \xrightarrow{j_1} u_1 \xrightarrow{j_2} \dots \xrightarrow{j_m} u_m$ with $j_1 < \dots < j_m$. As $\perp < j_1$ and $j_{i-1} < j_i$, Definition 1 creates the path $(u_0, \perp) \rightarrow (u_1, j_1) \rightarrow \dots \rightarrow (u_m, j_m)$ in G^{lev} .

(\Leftarrow) Any path $(s, \perp) \rightarrow (u_1, j_1) \rightarrow \dots \rightarrow (t, \ell)$ in G^{lev} projects to $s \xrightarrow{j_1} \dots \xrightarrow{j_m} t$ in G . Definition 1 ensures the edge conditions enforce $j_{i-1} < j_i$. ◀

This immediately yields the desired reduction: checking for a strictly increasing path in G reduces to a reachability query in G^{lev} .

Time Complexity. Using the bounds below, the construction of the levelled graph can be done in time polynomial in its input size.

► **Proposition 4.** One can construct G^{lev} from G by: (i) computing $L(v)$ for each node $v \in N$ (ii) creating a node (v, ℓ) for each $v \in N$ and $\ell \in L(v)$, and (iii) for each edge $u \xrightarrow{j} v$ in G and each $\ell \in L(u)$ with $\ell < j$, adding the edge $(u, \ell) \rightarrow (v, j)$. This can be done in $O(|N| + |E|^2)$ time.

Proof Sketch. Recall that $L(v) = \{j : \exists u, u \xrightarrow{j} v \in E\} \cup \{\perp\}$ hence we can compute $L(v)$ for all $v \in N$ in $O(|E|)$. For N' and E' the nodes and edges of G^{lev} , respectively, we have

$$|N'| = \sum_{v \in N} |L(v)| \leq \sum_{v \in N} (1 + \text{indegree}(v)) = |N| + |E|.$$

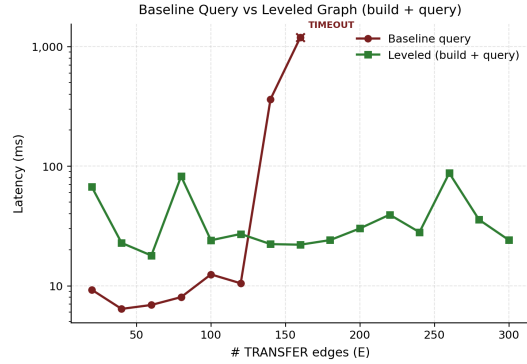
$$|E'| = \sum_{(u, j, v) \in E} |\{\ell \in L(u) \mid \ell < j\}| \leq \sum_{(u, j, v) \in E} |L(u)| \leq |E| \cdot \max_{u \in N} |L(u)| \leq |E|^2.$$

Hence, to compute G^{lev} we need $O(|E| + |N'| + |E'|) = O(|N| + |E|^2)$ time. ◀

26:4 Inexpressibility to Efficiency in Order-Constrained Paths

■ **Table 1** Baseline vs. leveled at $|N|=100$. Speedup is (baseline avg query)/(leveled build+leveled avg query).

$ E $	Leveled build (ms)	Leveled query (ms)	Baseline query (ms)	Speedup (\times)
20	57	10	9	0.13
40	17	6	6	0.26
60	12	6	7	0.39
80	76	6	8	0.10
100	16	8	12	0.50
120	20	7	10	0.37
140	15	7	363	16.50
160	15	7	1191	54.14
180	17	7	TIMEOUT	—
200	23	7	TIMEOUT	—
220	29	11	TIMEOUT	—
240	21	7	TIMEOUT	—
260	80	7	TIMEOUT	—
280	28	8	TIMEOUT	—
300	16	8	TIMEOUT	—



■ **Figure 2** Per-run average latency (log- y). Baseline rises and times out for $|E| \geq 180$ (10s limit), leveled query and build remains ≤ 100 ms across these sizes.

The increasing-path condition is expressible in the full language with set operations (in particular, difference) [4]. Here we deliberately work at the pattern-matching level because composing the required set operations leads to blow-ups, whereas compiling the order constraint into the leveled graph lets us answer it via plain reachability.

3 Experiments

We evaluate the leveled graph compilation for strictly increasing edge values by comparing: (i) a baseline query on the original graph that enforces $j_{i-1} < j_i$, and (ii) a leveled graph build along with a reachability query on the compiled leveled graph G^{lev} .

Setup. We use a local Neo4j instance accessed via the official Neo4j python driver.¹ The script rebuilds the base graph for each edge count, times the base build, compiles and times G^{lev} , and then runs repeated baseline/leveled queries with client-side per-run timeouts enforced by a worker process. The workload fixes $|N|=100$ accounts and varies $|E|$ from 20 to 300 in jumps of 20, each edge has an integer `amount` in $[1, 1000]$. We run 10 times per $|E|$ with `timeout=10` sec. Averages are over successful runs only.

Results. Leveled reachability is nearly flat across densities (~ 6 – 11 ms per run), whereas the baseline grows sharply and then times out. The leveled approach is slower for small graphs ($|E| \leq 120$), but becomes decisively faster as density grows: at $|E|=140$ we observe a **16.50 \times** speedup, rising to **54.14 \times** at $|E|=160$. For $|E| \geq 180$ all baseline runs hit the 10s per-run timeout, while leveled runs complete without timeouts. Note that results may vary with random graph instances, local resource contention and driver overhead.

¹ Experiments were run locally on macOS 15.6 (build 24G84) on a MacBook Air (Mac16,13) with an Apple M4 CPU (10 cores: 4 performance + 6 efficiency) and 16 GB RAM. The database was Neo4j 2025.07.1 (local install, default configuration), via the official Neo4j Python driver. Client stack: Python 3.13.5 and neo4j driver 5.28.2.

4 Conclusions

We revisited the expressiveness gap in core GQL and showed that compiling the order constraint into the graph reduces the inexpressible query to a standard reachability one while pointing to potential runtime benefits in our prototype implementation. Looking ahead, we propose to extend our method to address a broader class of queries (such as those including alternating order constraints) and to conduct a more systematic experimental study across different implementations, languages, datasets, and scales.

References

- 1 Marcelo Arenas, Claudio Gutiérrez, and Juan F. Sequeda. Querying in the age of graph databases and knowledge graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, pages 2821–2828, Virtual Event, China, 2021. Association for Computing Machinery. doi:10.1145/3448016.3457545.
- 2 Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. GPC: A pattern calculus for property graphs. In Floris Geerts, Hung Q. Ngo, and Stavros Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 241–250, Seattle, WA, USA, 2023. ACM. doi:10.1145/3584372.3588662.
- 3 Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. A researcher’s digest of GQL. In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPICs*, pages 1:1–1:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ICDT.2023.1.
- 4 Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. Gql and sql/pgq: Theoretical models and expressive power. *Proceedings of the VLDB Endowment (PVLDB)*, 18(6):1798–1810, 2025. Licensed under CC BY-NC-ND 4.0. doi:10.14778/3725688.3725707.
- 5 GQL Standards Committee. Gql standards website, 2024. Accessed: November 2024. URL: <https://www.gqlstandards.org/>.
- 6 Leonid Libkin, Wim Martens, Filip Murlak, Liat Peterfreund, and Domagoj Vrgoč. Querying graph data: Where we are and where to go. In *Companion of the 44th Symposium on Principles of Database Systems (PODS Companion '25)*, pages 1–19, Berlin, Germany, 2025. ACM. doi:10.1145/3722234.3725822.
- 7 Tobias Lindaaker. Predicates on sequences of edges. Technical report, ISO/IEC JTC1/ SC32 WG3:W26-027, 2023.
- 8 Hadar Rotschild and Liat Peterfreund. Order-Constrained-Paths-in-Leveled-Graphs. Software, swbId: `swb:1:dir:30effa7157a9e9bbf79b73b574a5189d5b5abef4` (visited on 2026-02-25). URL: <https://github.com/hadarrot/Order-Constrained-Paths-in-Leveled-Graphs>, doi:10.4230/artifacts.25284.
- 9 Sakshi Srivastava and Anil Kumar Singh. Fraud detection in the distributed graph database. *Cluster Computing*, 26(1):515–537, 2023. doi:10.1007/S10586-022-03540-3.
- 10 Fred Zemke. For each segment discussion. Technical report, ISO/IEC JTC1/ SC32 WG3:BGI-022, 2024.
- 11 Jun Zhu, Zaiqing Nie, Xiaojiang Liu, Bo Zhang, and Ji-Rong Wen. Statsnowball: A statistical approach to extracting entity relationships. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*, pages 101–110, Madrid, Spain, 2009. Association for Computing Machinery. doi:10.1145/1526709.1526724.