Meta-Property Graphs in Practice: Implementation and Evaluation

Stijn Nooijen
Eindhoven University of Technology
Eindhoven, The Netherlands
s.j.m.nooijen@gmail.com

Sepehr Sadoughi
Eindhoven University of Technology
Eindhoven, The Netherlands
s.sadoughi@tue.nl

Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
n.yakovets@tue.nl

Abstract

Property graph databases are now widely used to represent and store interconnected data, but they enforce a strict separation between data and metadata and do not support reification natively. This limits effective data management in those application areas where querying metadata and representing context are essential. The recently proposed Meta-Property Graph (MPG) model addresses both of these shortcomings through first-class treatment of labels and properties and formal mechanisms for reification; however, practical implementations in existing systems have not yet been considered. In this work, we address this gap: we present the design, implementation, and evaluation of a database engine extension that equips traditional property graph databases to support the MPG model. Specifically, we make four contributions: (1) we extend relational algebra to provide complete operational semantics for pattern-matching over MPGs; (2) we propose a query and storage engine architecture for processing MPG queries with MPG-specific optimizations; (3) we develop the first benchmark with a data generator and query suite targeting metadata-aware and reification-heavy workloads; and (4) we implement and validate our design in AvantGraph. Our evaluation confirms the practical feasibility of the MPG model and demonstrates its effectiveness on metadata-intensive workloads.

Keywords

Property Graphs, Reification, Metadata, Graph Databases, Database Design, Query Processing, Meta Property Graph

1 Introduction

Today, graph databases have become essential for managing complex, interconnected data across many diverse application domains such as social networks, knowledge graphs, biological networks, and others. Among graph data models, the property graph (PG) model stands out since ISO GQL (Graph Query Language) and SQL/PGQ offer an official ISO standard for it. In the PG model, nodes and edges carry labels and property name(key)/value pairs, providing an intuitive representation which aligns well with conceptual domain models.

Despite its success, PG data model imposes two fundamental limitations. First, it enforces a **strict separation between data and metadata**. Labels and property keys are *fixed* structural elements, not queryable data. Indeed, this rigidity surfaces in data integration, where something that constitutes data in one source may represent metadata in another, and in exploratory analytics, where the boundary between data and metadata shifts dynamically. Second, PG databases **lack native support for reification**: they cannot treat complex relationships or substructures as first-class queryable entities that themselves carry properties and relationships. Hence, representing provenance, temporal context, and other forms of metadata about data structures remains difficult and impractical.

The recently proposed Meta-Property Graph (MPG) model [10] addresses both of these limitations: it elevates labels and properties to queryable objects and introduces formal mechanisms for reification. However, the MPG model remains a purely theoretical construct with no practical implementation in an existing graph database system. Yet the applications in data governance, provenance tracking, and knowledge graph analytics increasingly demand the capabilities offered by MPG.

In this work, we bridge this theory-practice gap by presenting the design, implementation, and evaluation of a backwards-compatible database engine extension for MPG. Our main contributions are: (1) **Meta-GRA**, an extended graph relational algebra providing formal semantics for MPG pattern matching (§2); (2) **System architecture** for querying metadata and reified substructures with MPG-specific optimizations (§3); (3) **First MPG benchmark** with data generator and query suite for metadata-aware workloads (§4); (4) **Implementation in AvantGraph** with performance evaluation validating practical feasibility (§5). Together, these constitute the first concrete validation of the MPG vision.

2 Formal Foundation

The Meta-Property Graph (MPG) model [10] extends the traditional property graph model through two fundamental modifications: (1) while PG treats nodes and edges as primary objects and labels/properties as their metadata, MPG elevates labels and properties to first-class, identifiable objects alongside nodes and edges, enabling direct querying of any metadata in the graph; (2) MPG introduces reification, i.e., the ability to make statements about graph substructures, which may consist of a subset of nodes, edges, label sets, or properties.

Definition 2.1 (Meta-Property Graph). *A meta-property graph is a tuple* $G = (N, E, P, L, \lambda, \mu, \sigma, \nu, \eta, \rho)$ *where* $\mathcal{I}$, $\mathcal{L}$, $\mathcal{K}$, $\mathcal{V}$ *are pairwise disjoint sets of identifiers, labels, property keys, and values.* $N, E, P, L \subseteq \mathcal{I}$ *are finite sets of node, edge, property, and label set identifiers;* $\rho : E \to (N \times N)$ *maps edges to node pairs;* $\mu : L \to 2^{\mathcal{L}}$ *assigns labels to label set IDs;* $\lambda : (N \cup E) \to L$ *maps nodes/edges to label sets;* $\nu : P \to \mathcal{K} \times \mathcal{V}$ *assigns key-value pairs to properties;* $\sigma : N \cup E \to 2^{P}$ *assigns property sets to nodes/edges;* $\eta : N \to 2^{N \cup E \cup L \cup P}$ *assigns reified substructures to nodes (must be acyclic). Figure 1 illustrates an example MPG where node 8 reifies a substructure representing a reviewer assignment.*

2.1 Pattern Matching and Meta-GRA

Sadoughi et al. [9] introduced the full semantics of MetaGPML, extending GPML [2] with metadata-aware pattern matching. MetaGPML can bind metadata objects (labels and properties), navigate between data objects and their metadata (e.g., `(x:?y)` and metadata accessors), and express reification patterns where a

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.33

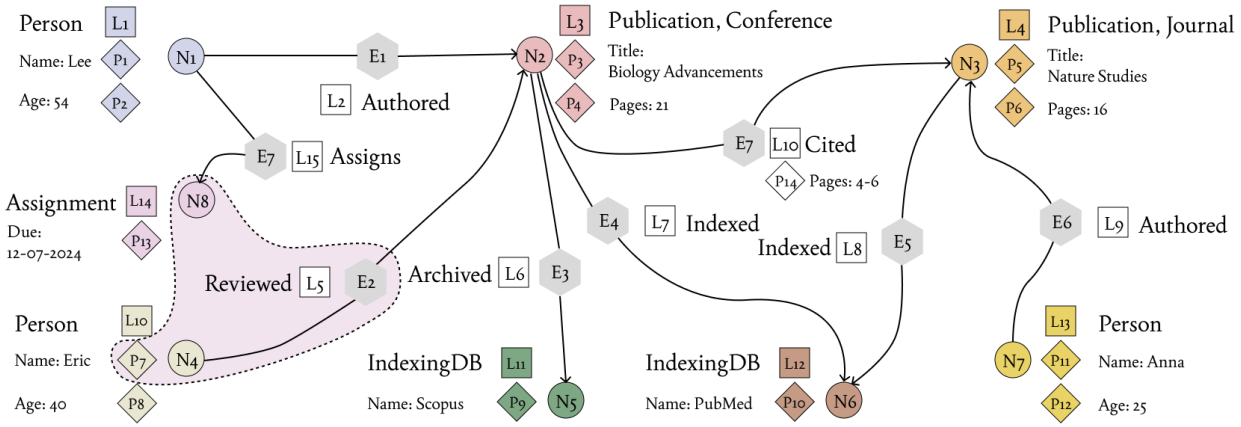


Figure 1: Example MPG modeling publications, index databases, and people. Unlike standard property graphs, properties (P) and label sets (L) have unique identifiers. Node 8 reifies a substructure containing edge 2, label set 5, node 4, and property 7, representing "Lee assigned Eric as a reviewer with deadline on 12 July 2024."

matched substructure becomes a first-class object (e.g., `(x::(y)-[]->(z))`). In short, metadata is now shifted from being merely annotations to being able to query targets that participate in pattern constraints.

To make these constructs executable, we define **Meta-GRA**: an extension of Graph Relational Algebra that makes label sets and property sets explicit. The goal here is twofold: (1) supply operational semantics for MetaGPML, and (2) permit equivalence-based query optimization [5]. Meta-GRA acts as the lowering target from patterns to algebraic plans. Once a MetaGPML query is expressed in Meta-GRA, it can be both executed and transformed using algebraic equivalences.

Core extensions. Meta-GRA adds operators that let queries observe metadata and iterate over metadata collections:

- $\pi_{\lambda(x)}$: project the label set identifier of object x
- $\pi_{\sigma(x)}$: project the set of property identifiers of object x
- $\omega_{y \Longrightarrow S}$: *unwind* a set-valued expression S into individual rows, binding each element to variable y
- $\bowtie_{x \in S}$: join predicate for testing set membership

The projection operators expose information that standard GRA treats implicitly. They show which labels apply to an object and which properties are present. The unwind operator ω flattens these set-valued attributes so that subsequent operators remain relational. A typical metadata-aware plan hence (i) projects a label/property set, (ii) unwinds it into rows, and (iii) applies membership constraints. The unwind operator is equally important for reification: it turns set-valued intermediate results into a flat relation, allowing a set-membership join to be rewritten as a standard equality join (which is important for good performance). Overall, Meta-GRA enables systematic, semantics-preserving transformation of MetaGPML constructs into algebraic plans, and it further provides the ability to reason about and optimize those plans uniformly, as demonstrated in our implementation (§3).

3 System Architecture

Here, we present a high-level design for extending a traditional PG database engine to support the MPG model, guided by four main principles: (1) **full backwards compatibility** with standard PG queries, (2) **zero-impact storage**, i.e., interpreting existing PG data as MPG without migration, (3) **maximal architectural reuse** of existing engine components, and (4) **minimal performance overhead** for queries not using MPG features.

3.1 Storage Model

Naturally, traditional PG engines assign identifiers only to nodes and edges. Making properties and label sets first-class citizens requires that they, too, be uniquely identifiable. For this, we employ **virtual identifiers** generated on-the-fly at query time. They are never stored persistently, satisfying the zero-impact storage constraint.

Virtual ID Generation. Since nodes and edges are typically stored in separate tables, their ID spaces may overlap (e.g., node ID 5 and edge ID 5 coexist). Virtual IDs must hence encode the object type to ensure global uniqueness:

- **Property IDs:** Generated from $\langle type, ownerID, key \rangle$. For example, the "Name" property of node 1 generates a unique ID distinct from a "Name" property on edge 1.
- **Label Set IDs:** Generated from $\langle type, ownerID \rangle$, ensuring label sets of nodes and edges with the same numeric ID remain distinct.

This encoding lets the engine reason about properties and label sets as if they had persistent IDs, and compatibility with existing storage layouts is maintained. The embedded owner ID points directly to the owning node or edge, enabling efficient lookups.

Reification Storage. Storing the set of object identifiers $\eta(n)$ that a node reifies requires a design choice. Here, we considered three options: (1) special edges from the reifying node to each object (rejected: edges cannot target other edges), (2) a separate reification table (rejected: requires expensive joins), and (3) storing reified IDs as list-valued properties. We adopt the third approach, encoding $\eta(n)$ in *hidden* columns. A single node-read retrieves all reified IDs without additional joins, reusing the existing table structure. Fig. 2 shows an example vertex table with four hidden columns storing reified node, edge, property, and label set IDs.

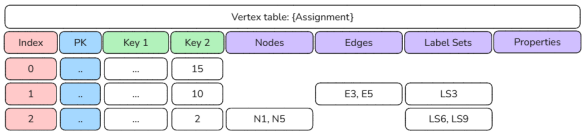


Figure 2: Example vertex table storage with reification columns. Four hidden columns store lists of reified object IDs (nodes, edges, properties, and label sets) for each reifying node.

3.2 Query Processing Architecture

Our MPG extension introduces several novel rewrite rules during query planning. These rules transform MPG operations into sequences of existing PG operators. This maximizes code reuse and leverages the base engine's existing optimizations.

Metadata Awareness. Queries for all label sets or properties must now scan both nodes and edges. Since these reside in separate tables, we rewrite global metadata scans into UNIONs:

- Query `|ls|` (all label sets) rewrites to `(:?ls)` ∪ `()-[:?ls]->()`:

$$\pi_{ls=\lambda(x)}(\bigcirc_x) \quad \cup \quad \pi_{ls=\lambda(xy)}(\uparrow_x^y (\bigcirc_x))$$

- Query `{p}` (all properties) requires gathering property sets from both nodes and edges, then unwinding to individual properties:

$$R_1 = \pi_{p_1=\sigma(x)}(\bigcirc_x) \cup \pi_{p_1=\sigma(xy)}(\uparrow_x^y (\bigcirc_x))$$

$$R_2 = \omega_{p_2 \Longrightarrow p_1}(R_1)$$

Reification Processing. When a query accesses a reified substructure, the engine needs to expand $\eta(n)$ into individual objects. After parsing, join conditions containing set-membership tests ($y \in \eta(x)$) are rewritten using the unwind operator:

$$\bigcirc_x \bowtie_{y \in \eta(x)} \bigcirc_y \quad \Rightarrow \quad \omega_{m \Longrightarrow \eta(x)}(\bigcirc_x) \bowtie_{y=m} \bigcirc_y$$

This transforms set-membership joins into standard equality joins over unwound data, allowing the base engine's existing join algorithms (nested-loop, hash, index-based) to be reused directly.

3.3 MPG-Specific Optimizations

Filter pushdown: A key optimization is **filter pushdown for properties and labels**. A naive planner applies filters like `KEY(p) = "name"` only after unwinding and joining. Instead, we push such filters down immediately after unwinding, before the expensive joins. Specifically, label filters are pushed into the access operation so that only matching vertex tables are scanned. Property key filters are pushed into the property-read operation, which accepts an optional key list and retrieves only the requested columns rather than all properties. The optimizer also handles disjunctive key predicates (`OR`/`IN`) by pushing a combined key list, and detects conflicting conjunctive key predicates to eliminate unreachable sub-plans entirely. Our evaluation (§5.2) confirms up to 59.0% speedup on queries that benefit from this rewrite.

Cost-Based Planning: MPG workloads also benefit from **cost-based planning choices** that are less prominent in traditional PG workloads. Specifically, the optimal operator order depends on whether it is cheaper to resolve reification early ("reification-first") or to traverse first and connect results via reification afterwards ("traversal-first"). A small change in predicate selectivity can significantly affect the optimal plan: a highly selective filter on a reifying structure favors early reification resolution, while a highly selective filter on a traversed endpoint favors starting from that endpoint and performing reverse reification lookups only for the reduced candidate set. This motivates the need to use a cost-based optimizer which uses graph's data/metadata statistics (e.g., cardinalities, selectivities, and reification density) to pick between lowest-cost alternatives rather than relying on fixed rewrite rules.

Reverse Reification Index: Storing reified object identifiers as lists on the reifying node is efficient for forward lookups ("given a reifying node, retrieve its reified objects") but makes reverse lookups ("which node reifies this object?") quite expensive: the engine needs to scan all potential reifying nodes and test list membership. An optional specialized index can invert these references. It maps each object identifier to the reifying node(s) that mention it, turning reverse reification from a scan into an indexed lookup. This introduces a trade-off: storage footprint and write-time maintenance overhead increase (e.g., updates on insert/delete); hence the index is best employed for workloads dominated by reverse reification patterns. Beyond query performance, the index also helps with referential integrity maintenance: when a graph object is deleted, the engine can use the index to locate and clean up all reifying nodes that reference it, avoiding a costly full re-scan.

Overhead for Standard PG Queries. By design, standard property graph queries execute without overhead. Since the parser identifies MPG constructs at compile time, standard queries hence incur no additional runtime checks or plan rewrites. MPG extensions activate only when MPG-specific syntax appears (metadata binding, reification patterns); all other queries follow the original execution path unchanged. This satisfies our principle of minimal performance overhead.

4 Benchmark Design

Existing graph database benchmarks, including the LDBC Social Network Benchmark (SNB) [11], cover traditional property graph workloads well. However, they cannot evaluate MPG-specific features: no existing benchmark treats metadata as first-class objects or exercises reification. To address this gap, we developed the first benchmark specifically targeting metadata-aware and reification-heavy workloads.

Design Requirements. An effective MPG benchmark must satisfy four key requirements: (1) stress metadata-as-data queries that bind label sets and properties to variables and iterate over metadata collections; (2) comprehensively test reification through forward lookups (retrieving substructures from reifying nodes), reverse lookups (finding which nodes reify a given object), complex multi-element pattern matching, nested reification, and overlapping reifications; (3) provide realistic, scalable data with tunable MPG-specific parameters such as reification density; and (4) build on well-established benchmarks for credibility and comparability.

Foundation: LDBC SNB BI. To achieve these goals, we have chosen to extend the LDBC SNB Business Intelligence workload. SNB models a realistic social network with entities like `Person`, `Post`, `Comment`, and relationships like `knows` and `likes`. The BI workload features complex, aggregation-heavy analytical queries designed using LDBC's "choke point" methodology to expose database bottlenecks. Its data generator produces scaled, correlated, and skewed data, essential for realistic performance

evaluation. The rich schema offers context for defining meaningful metadata queries. The established credibility of SNB (in the community) ensures the results are comparable and trustworthy.

MPG features also affect write paths (insert, delete, update), but our benchmark focuses deliberately on read-only queries. This is because MetaGPML specifies pattern-matching semantics but *does not* define DML for updates; hence our prototype targets query execution rather than transactional maintenance. In a write-capable MPG engine, the main additional costs would stem from maintaining MPG-specific state under modifications: updating reification lists on insert/change and ensuring deletes do not leave dangling references. We leave a systematic study of write semantics for future work. Nevertheless, to reflect write-side costs, we report the initial graph loading time across different scale factor variants. This captures the one-time overhead of constructing MPG-specific structures during ingestion (e.g., reification columns and encoded identifier lists).

Data Generation Extension. We extend the SNB schema to support reification. `Message` nodes (Posts and Comments) can now reify substructures representing their semantic content. For example, rather than storing "John started working at my university" as unstructured text, a message can reify the actual `Person` node for John and the `workAt` edge connecting him to a University.

Our data generator extension employs multiple *populators*, each generating specific types of reifiable content: complete `Person` nodes; `knows`, `isLocatedIn`, and `studyAt` relationships; labels from `Organisation` nodes and `replyOf` edges; `content` properties from messages; and `workFrom` from `workAt` edges.

For each message, the generator probabilistically selects a (possibly empty) set of populators, multiple populators may also be selected for a single message. This creates complex, multi-element substructures that stress diverse reification patterns. Three parameters are used to control the generation: (1) the probability that any message attempts reification (default: 25%), (2) the probability that each populator is selected when reification occurs (default: 10%), and (3) each selected populator adds 0–n (default: 10) randomly chosen elements, maintaining set semantics to avoid duplicates.

Query Suite. Our benchmark includes 12 new queries specifically designed to stress MPG capabilities, covering metadata-aware and reification-based pattern matching: queries retrieving all label sets and properties, finding properties with specific keys, forward lookups retrieving substructures from reifying nodes, reverse lookups finding reifiers of given objects, complex pattern matching over multi-element substructures, nested reification (nodes reifying other reifying nodes), and queries finding overlapping reifications where multiple nodes share reified elements. These query patterns reflect realistic scenarios including provenance tracking (forward/reverse reification lookups), schema introspection (metadata binding), and knowledge graph analytics with qualifiers (nested and overlapping reification).

5 Implementation and Evaluation

To validate our design, we implemented the proposed MPG extension in AvantGraph [12], our graph query processing engine which is being developed in TU Eindhoven. Below, we describe key implementation aspects and present experimental results confirming practical feasibility.

5.1 Implementation in AvantGraph

Storage Model. AvantGraph employs a partitioned storage architecture. A vertex catalogue maps each unique label set to a dedicated vertex table. Each table stores only nodes with identical labels and represents properties as columns. Homogeneous partitioning yields efficient column-oriented access. An edge catalogue maps combinations of (source label set, edge label, target label set) to dedicated edge tables. Edges are stored bidirectionally, enabling efficient traversal in either direction without reverse index scans.

Virtual ID Generation. Properties and label sets lack physical IDs in the original storage model, hence we generate virtual IDs on-the-fly using composite keys. A label set ID consists of (table ID, vertex ID, edge offset); edge offset is null for node label sets. A property ID extends this with a column ID: (table ID, vertex ID, edge offset, column ID). These composites let properties and label sets act as first-class objects; the underlying storage format remains unchanged.

Virtual identifiers also introduce two non-obvious engineering challenges. First, they **force a strict separation between identity and data access**. In a traditional engine, a variable bound to a vertex/edge implies a fully materialized element. Reification, however, stores references (IDs) rather than duplicated elements. Treating these references as full elements would either allow accidental "read-through" without an explicit lookup or trigger implicit, repeated fetches. The engine hence represents reified results as ID-only values. An explicit materialization step runs only when later operators need labels/properties. Second, **virtual IDs complicate physical lowering and joins**. Once lowered, an abstract ID becomes a tuple of physical columns. Our unified encoding for label sets uses a null component to distinguish node-owned from edge-owned cases. A naïve equality join would miss correct matches under standard null semantics; joins over lowered identifier components must instead use null-safe equality, to preserve the intended matching behavior.

Reification Storage. We add four columns to each vertex table, one per reified object type (nodes, edges, properties, label sets; see Fig. 2). Each column carries a metadata flag distinguishing it from regular property columns. Object IDs within each list use radix encoding (Fig. 3), which handles variable-length sequences efficiently. Control bytes indicate null values and list continuation, enabling compact storage while maintaining proper ordering for database operations. During graph import, the engine builds a translation map from external IDs to internal physical IDs, validates acyclicity of all reification references via a transitive closure check, and then radix-encodes the translated ID lists into the hidden columns.

Query Parser. AvantGraph's ANTLR-based Cypher parser is extended with MetaGPML syntax: (1) `:?var` for binding label sets, (2) `..var` for binding properties (double-dot avoids ambiguity with existing single-dot syntax), (3) `|var|` and `{var}` for matching all label sets and properties, and (4) `::pattern` for reification queries. Nested reification queries like `(a::(b::(c)))` employ a stack-based parsing strategy that tracks named variables within each reification layer, producing correct join conditions ($c \in \eta(b)$, $b \in \eta(a)$). This flattening of nested reification into explicit joins is analogous to the decorrelation of correlated subqueries in relational engines, enabling the optimizer to select efficient join algorithms rather than resorting to iterative nested-loop evaluation.

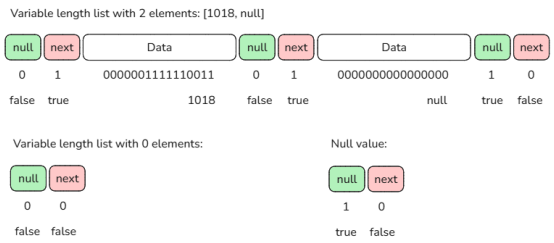


Figure 3: Radix encoding for variable-length reification lists. Control bytes (green/red) indicate null values and list continuation. Examples show encoding of [1018, null], empty list, and null value.

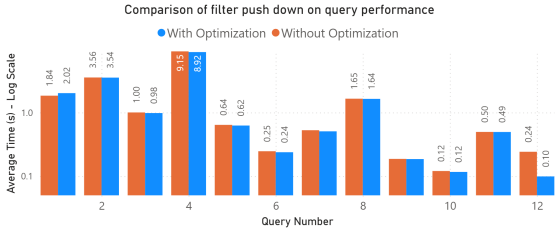


Figure 4: Comparison of average query execution time for each query in the benchmark suite, with and without the filter pushdown optimization. Y-axis is logarithmic scale; lower is better. Query 12 shows 59.0% improvement.

Query Execution Engine. During query planning, we apply the rewrite rules from §3. Metadata scans become UNIONs over node and edge scans; set-membership joins ($y \in \eta(x)$) are transformed via unwind operators into equality joins. Further, we inject filter pushdown immediately after unwind operations, reducing data volume before expensive joins. A current limitation is heterogeneous property types: AvantGraph enforces strict column typing, so the property-read materialization handles only string-valued properties. Supporting mixed-type property lists (strings, integers, dates) would require union types in the intermediate representation, which we leave for future work.

5.2 Experimental Setup

Environment. All experiments ran on a GitHub Codespaces VM with 16 cores (AMD EPYC 7763), 64GB RAM, and 128GB NVMe SSD, running Ubuntu 24.04.

Dataset and Metrics. We generated SNB data at Scale Factor 0.3 (1.06 GB on disk) with reification parameters: 25% of messages attempt reification, each populator has 10% selection probability, 1–10 elements per selected populator. The query suite comprises 12 analytical queries targeting metadata-aware and reification-based pattern matching. We measured end-to-end query latency, averaging 30 runs per query, comparing two configurations: baseline (no MPG-specific optimizations) and optimized (filter pushdown for properties and labels enabled).

5.3 Results

Optimization Impact. Fig. 4 shows that filter pushdown significantly improves performance on queries with selective metadata filters. Query 12, which filters properties by key predicates, achieves a 59.0% speedup. Other MPG queries show minimal overhead (<3%), confirming that the optimization does not penalize queries that do not benefit from it.

Scalability Analysis. We conducted three experiments to evaluate how performance scales with reification characteristics. Each experiment varied a single parameter while holding others constant (25% reification frequency, 10% populator probability, max 10 elements).

Parameter Sensitivity. Fig. 5 shows how query performance varies with three key parameters. Substructure size (a) has the largest impact. Queries 1, 2, 3, 8, 11 degrade linearly as max elements grows from 1 to 50. Reification frequency (b) has moderate impact across the 1%–100% range. Type diversity (c) has the least impact, since the engine partitions storage by object type. A notable outlier is Query 4, which maintains a stable ~10 s execution time regardless of parameter changes. Its highly selective geographic and temporal filters reduce the candidate set before reification processing begins, making it insensitive to reification density.

Comparative Analysis. Fig. 6a presents a normalized comparison across all three parameters. Substructure size is the most influential factor (519% degradation), followed by type diversity (433%); reification frequency has the least relative impact (191%). The engine is indeed more sensitive to processing complex individual substructures than to discovering larger numbers of simpler ones. The **reverse reification index** proposed in §3 can substantially reduce this degradation, specifically for queries with highly selective predicates applied within a substructure (queries 2, 4, 5).

Graph Load Time. To evaluate ingestion overhead, we define four reification density categories (dense, large, medium, and small). They vary in maximum substructure size, reification frequency, and type diversity (Table 1).

Table 1: Parameters for reification density categories.

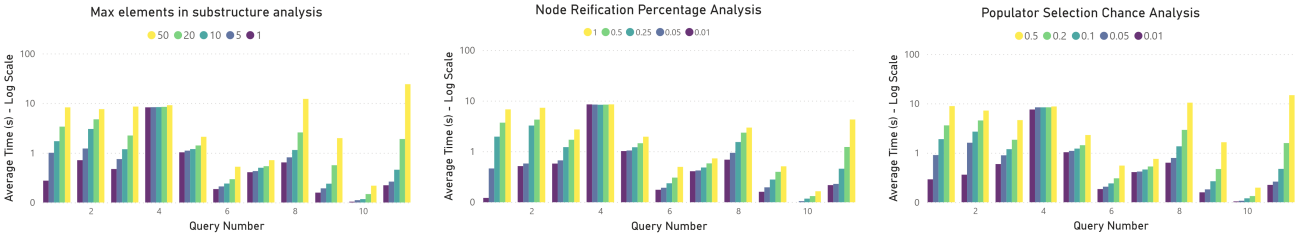
Category	Max elements	Frequency	Diversity
dense	20	0.50	0.20
large	10	0.25	0.10
medium	5	0.05	0.05
small	1	0.01	0.01

Fig. 6(b,c) presents two additional scalability experiments. Load times (b) remain stable at lower scale factors. They increase significantly at SF 0.3 as density grows. Query execution times (c) trend upward consistently with scale factor. At SF 1.0, several queries exceed 10 seconds. The gap between density categories widens at larger scale factors, reflecting the growing cost of encoding and indexing reification lists during ingestion.

Our results confirm three key findings: (1) MPG capabilities can be realized in existing PG engines without data migration; (2) MPG-specific optimizations yield significant performance gains on reification-heavy queries; (3) performance scales reasonably with reification density and complexity. Together, these findings validate the practical feasibility of our architectural design. They further highlight that targeted optimizations such as filter pushdown and the reverse reification index become increasingly important as reification density grows.

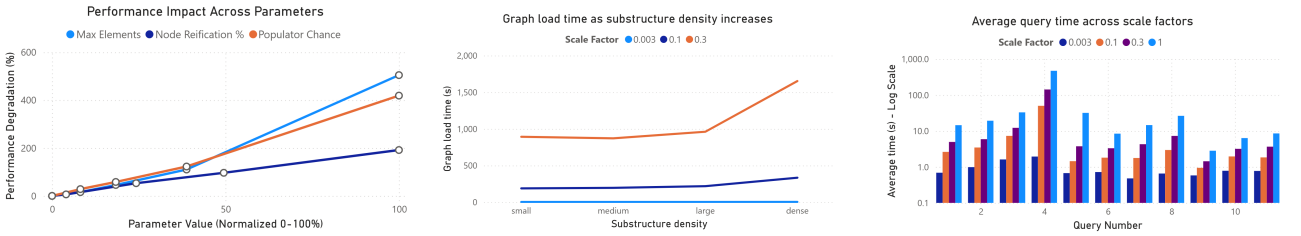
6 Related Work

Data-Metadata Heterogeneity. Flexible data-metadata management has been studied in relational contexts: SchemaSQL [6] treats relation/attribute names as queryable, FISQL [13] adds



(a) Max elements **(b) Reification %** **(c) Type diversity**

Figure 5: Impact of data generation parameters on query performance (log scale).



(a) Parameter impact **(b) Load time vs. density** **(c) Query time vs. scale factor**

Figure 6: Scalability analysis: (a) normalized performance degradation across data generation parameters, (b) graph load time across reification densities, (c) query execution time across scale factors 0.003–1.0 (log scale).

dynamic output schemas, and Hernandez et al. [4] address data-metadata translation. Our work extends these principles to property graphs.

Graph Models and Reification. RDF and RDF-star support reification via named graphs and quoted triples but sacrifice the conceptual clarity of property graphs. Statement Graphs [3] and OneGraph [7] bridge RDF and PG yet require entirely new models and query languages. Our approach instead maintains full ISO GQL/SQL-PGQ compatibility while adding metadata-awareness and reification as backwards-compatible extensions. Meta-GRA extends graph relational algebra [5, 8] with metadata and reification operators. While LDBC SNB [11] covers PG benchmarking comprehensively, it lacks metadata-awareness; our benchmark is the first to target these capabilities.

Position. To our knowledge, this is the first implementation and evaluation of a metadata-aware PG system with reification, providing practical validation of the MPG vision [10].

7 Conclusions and Future Work

In this work, we presented the first practical implementation of the recently proposed Meta-Property Graph model. The backward-compatible design demonstrates that metadata awareness and reification do not require the migration to an entirely new data model or query language. Existing property graph databases can adopt these capabilities while preserving full compatibility with existing queries and data. Our contributions span Meta-GRA for operational semantics, a storage architecture using virtual IDs and reification lists, a number of novel query optimizations, the first MPG benchmark extending LDBC SNB, and a validated implementation in AvantGraph. Future work includes cost-based optimization and reverse reification indexes, deploying our approach in real-world applications in data governance and provenance, model extensions building on PG-SCHEMA [1], and finally DML operations.

8 Artifacts

The GitHub repository of the developed benchmark for metadata-aware and reification-heavy workloads is available at:
https://github.com/StijnKing/ldbc_snb_bi

Acknowledgments

This work was supported by the MATTER-TKI-HTSM/22.0024 research grant.

References

[1] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, Michael Schmidt, Juan Sequeda, Sławek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Dušan Živković. 2023. PG-Schema: Schemas for Property Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 198:1–198:25. doi:10.1145/3589778

[2] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, 2246–2258. doi:10.1145/3514221.3526057

[3] Ewout Gelling, George Fletcher, and Michael Schmidt. 2024. Statement Graphs: Unifying the Graph Data Model Landscape. In *Database Systems for Advanced Applications - 29th International Conference, DASFAA 2024*. 364–376.

[4] Mauricio A. Hernández, Paolo Papotti, and Wang-Chiew Tan. 2008. Data Exchange with Data-Metadata Translations. *Proceedings of the VLDB Endowment* 1, 1 (2008), 260–273. doi:10.14778/1453856.1453888

[5] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j.. In *EDBT/ICDT Workshops (CEUR Workshop Proceedings, Vol. 1558)*, Themis Palpanas and Kostas Stefanidis (Eds.). CEUR-WS.org. http://dblp.uni-trier.de/db/conf/edbt/edbtw2016.html#HolschG16

[6] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. 2001. SchemaSQL: An Extension to SQL for Multidatabase Interoperability. *ACM Transactions on Database Systems* 26, 4 (2001), 476–519. doi:10.1145/503099.503102

[7] O. Lassila, M. Schmidt, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, R. Sharda, and B. Thompson. 2022. The OneGraph Vision: Challenges of Breaking the Graph Model Lock-In. *Semantic Web Journal* 14, 1 (2022), 125–134. doi:10.3233/SW-223273

[8] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. Formalising openCypher Graph Queries in Relational Algebra. In *Advances in Databases and Information Systems*. Springer International Publishing, 182–196.

[9] Sepehr Sadoughi, Nikolay Yakovets, and George Fletcher. 2024. Meta-Property Graphs: Extending Property Graphs with Metadata Awareness and Reification. *arXiv preprint arXiv:2410.13813* (2024). doi:10.48550/arXiv.2410.13813

[10] Sepehr Sadoughi, Nikolay Yakovets, and George HL Fletcher. 2025. Breaking Down the Data-Metadata Barrier for Effective Property Graph Data Management. In *28th International Conference on Extending Database Technology, EDBT 2025* (Barcelona, Spain) *(EDBT '25)*. OpenProceedings.org. doi:10.48786/edbt.2025.80

[11] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC social network benchmark: Business intelligence workload. *Proceedings of the VLDB Endowment* 16, 4 (2022), 877–890.

[12] Wilco van Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. AvantGraph Query Processing Engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3698–3701. doi:10.14778/3554821.3554878

[13] Catharine M. Wyss and Edward L. Robertson. 2005. Relational Languages for Metadata Integration. *ACM Transactions on Database Systems* 30, 2 (2005), 624–660. doi:10.1145/1071610.1071618