TurboLynx: Schemaless Graph Engine Strikes Back for General-Purpose Analytics

Taesung Lee
POSTECH
Pohang, Republic of Korea
tslee@dblab.postech.ac.kr

Jaehyun Ha
POSTECH
Pohang, Republic of Korea
jhha@dblab.postech.ac.kr

Byungchul Tak*
Kyungpook National University
Daegu, Republic of Korea
bctak@knu.ac.kr

Wook-Shin Han*
Graduate School of AI, POSTECH
Pohang, Republic of Korea
wshan@dblab.postech.ac.kr

ABSTRACT

Graph database management systems (GDBMSes) are widely adopted for their efficient handling of graph traversal queries that capture complex relationships. Recently, a class of modern GDBMSes appeared that were designed to offer explicit support for *schemaless* property graph models (PGMs), providing users with a high degree of flexibility. However, GDBMSes in this class often suffer from performance bottlenecks in analytical database queries—typically involving operations such as group-by and aggregation. We argue that a major cause is that schemaless processing is not treated as a primary design requirement across the storage, query-processing, and optimization layers. To address this, we propose TurboLynx, a novel graph analytics engine that holistically integrates the schemaless property at every layer of the system—from storage to query processing and optimization. TurboLynx organizes graph data into cost-based clusters, called graphlets, and stores them in a columnar format. By adopting a graphlet-aware query optimizer and processor, TurboLynx efficiently handles both graph traversal and analytical workloads in a single system. Our comprehensive evaluation on `LDBC SNB Interactive`, `TPC-H`, and `DBpedia` demonstrates that TurboLynx outperforms state-of-the-art GDBMSes by up to 183.9× and leading RDBMSs by up to 41.27×.

PVLDB Reference Format:
Taesung Lee, Jaehyun Ha, Byungchul Tak, and Wook-Shin Han. TurboLynx: Schemaless Graph Engine Strikes Back for General-Purpose Analytics. PVLDB, 19(6): 1250 - 1263, 2026.
doi:10.14778/3797919.3797932

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/postechdblab/TurboLynx.

1 INTRODUCTION

Graph database management systems (GDBMSes) offer advantages over conventional RDBMSes by enabling natural and efficient handling of highly connected data and relationship-centric queries, including path expansion, variable-length pathfinding, and shortest-path computation. Among various graph data models, the *property graph data model* (PGM) has emerged as one of the most popular approaches [27, 37, 74], often preferred over graph data models such as Resource Description Framework (RDF) [62, 68]. It is supported by major GDBMSes, such as Neo4j [49], TigerGraph [66], and Amazon Neptune [50]. In the PGM, edges and vertices can be labeled and have an arbitrary number of associated key-value pair attributes. Since these attributes need not be predefined, the model is often referred to as *schemaless* [34, 41, 67, 70, 72]. This flexibility is particularly beneficial in agile development environments and in scenarios where the data model is not fully understood at the outset. However, GDBMSes supporting schemaless PGMs often experience significant performance bottlenecks in general analytics, such as group-by and aggregation [16]. We attribute these bottlenecks to the fact that schemaless processing is not treated as a primary design requirement across the storage, query processing, and optimization layers.

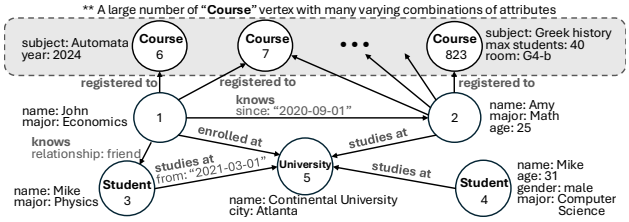


Figure 1: An example data following PGM.

Concretely, we identify three main reasons why performance degrades in schemaless PGM-supporting GDBMSes when performing such complex analytic queries, each corresponding to a missing integration of schemaless processing in the storage structure, the query processing engine, or the query optimizer. First, regarding *storage structure*, *native* GDBMSes (e.g., Neo4j and Memgraph [46]) adopt a *row-based* layout that embeds schema information directly into each tuple. While this approach provides great flexibility by allowing each tuple to hold distinct attributes, it also leads to considerable runtime overhead due to the need for per-tuple schema interpretation. Second, in *query processing*, because data with varying schemas is intermixed, these systems tend to rely on Volcano-style operators without vectorization. Even Neo4j's batched processing [56] cannot fully exploit `SIMD`-based vectorization. Fixed-schema stores—as used by engines like Kuzu—can easily apply the vectorization, whereas schemaless stores struggle as they do not natively maintain a columnar layout or a fixed record layout; consequently, physical operators in graph database engines that process schemaless data have difficulty in leveraging `SIMD` for vectorization. Third, at the *query optimization* stage, most native GDBMSes lack advanced optimization techniques in their custom-built optimizers. For instance, Neo4j uses heuristics for cardinality estimates (covering equality, range predicates, and distinct

*Corresponding authors

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797932

operations) and has only limited memoization for join ordering. Moreover, it includes only about 20% as many query rewrite rules as state-of-the-art optimizers such as Orca [60] and Calcite [10].

An alternative to building a native GDBMS from scratch is the *build-on-top* approach, which layers a graph interface over an existing RDBMS engine [29, 61, 62, 65]. Graph queries are translated into SQL and executed on relational data, thereby reusing mature RDBMS components—storage engines and query compilers among them. However, it usually does not offer the same degree of schema flexibility since this method depends on an RDBMS [40, 65]. Furthermore, such *build-on-top* GDBMSes either struggle with extremely sparse tables (i.e., many null columns) or must manage a multitude of distinct schemas. Conversely, *native* GDBMSes implement graph functionality from the ground up [46, 49, 66], delivering deeper integration and faster traversals. However, achieving a level of maturity comparable to established RDBMS query optimizers requires substantial effort and time. Consequently, we face a dilemma: build a native GDBMS to maximize graph-traversal performance but risk slower, less-optimized analytics, or build on top of an RDBMS and sacrifice some schema flexibility and graph-traversal performance.

To address this dilemma, we propose a new graph analytics system called TurboLynx, designed to handle schemaless data end-to-end—encompassing storage, query processing, and query optimization. TurboLynx integrates several techniques to tackle the aforementioned challenges. At the storage layer, TurboLynx employs a *columnar* format for PGM data and groups similar schemas into coarse-grained units, called *graphlets*, via a cost-based merging process. This design obviates per-tuple schema interpretation, enables vectorized execution, and speeds up attribute accesses. In addition, our system can accelerate queries that filter on specific attributes by leveraging intersection and union operations on *inverted indexes* built over metadata (i.e., attribute names).

For query processing, TurboLynx extends each operator to enable handling of vertices and edges from multiple graphlets. For instance, the Cypher query `MATCH (n:Course)` in Figure 1 is executed via a scan operator over multiple graphlets because `Course`-labeled nodes vary significantly in their attributes. This scenario poses two main challenges: (1) operators must process inputs spanning multiple schemas, and (2) each binary join between vertices labeled R and edges labeled S can lead to a combinatorial explosion of intermediate schemas, proportional to $|\mathcal{N}(R)| \times |\mathcal{E}(S)|$ *. To address the first challenge, TurboLynx enhances each operator to natively consume multi-schema inputs. For the second challenge, we introduce the *Shared Schema Row Format (SSRF)* with a *unified header* scheme. By storing schema definitions separately from the actual data, *SSRF* reduces schema management overhead and maintains low null-data handling costs. *SSRF* also selectively combines columnar and row-based layouts—attributes that are mostly null reside in a row-based format—so that each row references a *shared schema metadata* entry rather than embedding a full schema. Collectively, these techniques mitigate both the overhead of null values and the combinatorial blowup of managing many intermediate schemas.

At the query optimization layer, TurboLynx adapts the Orca [60] optimizer, originally developed for relational databases, to leverage its established optimization techniques. TurboLynx augments Orca with graph-specific cost models, rules, and operators to accommodate its graphlet-based design. However, treating each graphlet as a separate table can cause the plan search space to grow overwhelmingly large. To address this, we propose an "early merging" strategy based on the Greedy Operator Ordering [25] that merges graphlets before join enumeration, thereby reducing the overall plan search space to manageable levels while still preserving graphlet flexibility.

In addition, like other analytical engines such as ClickHouse [57] and Apache Druid [71], TurboLynx supports batch updates, making it well-suited for big data analytics scenarios that prioritize high-throughput read performance while consolidating updates in bulk. Just as Delta Lake [8] provides transactional guarantees for Spark, TurboLynx could be extended to offer similar capabilities; however, exploring this potential is out of our scope.

We make the following contributions. First, focusing on GDBMSes with schemaless graph data support, we identify inadequate handling of schemaless graph data as a primary cause of underperformance across storage, execution, and query optimization. Based on this, we present the key design factors of a novel graph analytics system that overcomes this problem. Our design comprehensively covers the storage layer, query processing engine, and query optimizer to deliver a drastically improved performance. Second, we provide a fully functional prototype, TurboLynx, that implements the proposed design. Third, we demonstrate the efficacy of TurboLynx through comprehensive evaluations against diverse systems.

2 BACKGROUND

2.1 Property Graph Model

We formally define the property graph model [4, 6, 9, 11, 18, 27, 28, 38]. A property graph is a tuple $G = \{N, E, L, K, V, \rho, \lambda, \sigma\}$ where N and E are finite sets of nodes and edges, respectively, and L, K, V are sets of labels, property keys, and property values, respectively. The functions are as follows:

- $\rho : E \rightarrow (N \times N)$ is a function that maps edges to ordered pairs of nodes
- $\lambda : (N \cup E) \rightarrow \mathcal{P}(L)$ is a function that maps nodes or edges to their associated label set, where $\mathcal{P}(L)$ is the power set of L
- $\sigma : (N \cup E) \times K \rightarrow V$ is a function that maps a property key of a node or edge to a property value

2.2 Query Language for PGM

TurboLynx supports Cypher [27] as the query language for PGM, since it is the official and widely adopted query language for Neo4j, a leading GDBMS, and provides strong support for *graph pattern matching* as its primary feature. This focus on pattern matching not only makes Cypher effective for expressing complex graph queries, but also facilitates understanding and potential extension towards the international standard ISO/GQL [36], which is also heavily based on graph pattern matching principles. Although other influential languages predating ISO/GQL, such as Gremlin [55], GSQL [20], and G-Core [5], also utilize graph pattern matching, Cypher serves as a clear and illustrative example of this approach. For example, the query in Step 2 of Figure 2 finds people whose first name starts with 'Fran' and who live in cities with an area greater than 100, returning both the person and the city.

*We define $\mathcal{N}(R)/\mathcal{E}(S)$ for the set of vertexlets/edgelets labeled R/S in §4.1.2, where "vertexlets/edgelets" are graphlets storing vertices/edges, respectively.

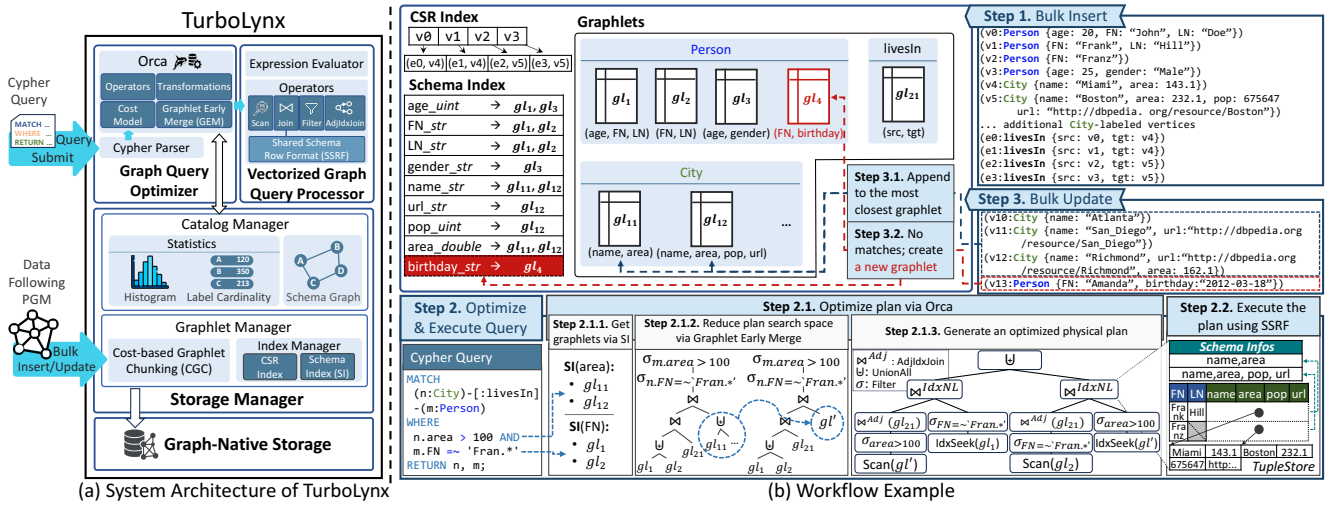


Figure 2: Overview of TurboLynx architecture illustrating interactions between various components and a workflow example.

2.3 Storage Architectures for PGM

Figure 3 shows sample input data and Neo4j's storage architecture, focusing on how node properties are organized. It stores nodes, relationships, and properties in fixed-size records for offset-based access. Each node record holds its unique ID, associated label(s), and pointers to its first property and relationship records. A property record has up to four 8-byte "property blocks," each encoding a property key (e.g., age), its data type, and the corresponding value. Variable-length values (e.g., strings) may span multiple consecutive blocks. Node properties form a doubly linked list of property records, allowing a variable number of properties per node. A relationship record stores pointers to its start and end nodes and links to the previous and next relationship records in each node's adjacency list, forming bidirectional linked lists.

Memgraph is an in-memory GDBMS that, unlike Neo4j's fixed-size record approach, stores node and edge properties in variable-size records. As in Neo4j, each record embeds its schema information, so both systems track metadata alongside data values.

2.4 Query Optimization for PGM

Neo4j employs a cost-based optimizer based on dynamic programming with a memo table. It relies on simple statistics [43] (e.g., node/edge counts per label) and heuristic selectivities (e.g., 0.05 for equality). Lacking attribute-level metadata (e.g., which attributes exist, their types, or average widths), the optimizer cannot model tuple sizes and thus often misestimates I/O and memory costs.

Memgraph uses an even simpler custom cost-based optimizer than Neo4j's and likewise lacks attribute-level statistics. Join ordering begins by selecting a single start node from the query graph and then performing a random walk along connected edges, incrementally extending the plan. This strategy naturally produces only *left-deep* join trees and fails to explore bushy or zig-zag alternatives, which can lead to markedly sub-optimal plans in complex queries.

3 OVERVIEW OF TURBOLYNX

Figure 2 presents TurboLynx's architectural overview (Figure 2a) with a representative workflow (Figure 2b) that spans bulk ingestion, query optimization/execution, and bulk updates.

```
(v0:Person {ID:0, age:20, FN:"John", LN:"Doe", Job:"Chef"})
(v1:Person {ID:1, FN:"Frank",  LN:"Hill"})
(v2:Person {ID:2, FN:"Franz"})
```

- FN: FirstName
- LN: LastName

(a) Input vertices data

next	prev				
●	●	ID(uint):0	Age(int):20	FN(str):"John"	LN(str):"Doe"
↓	●	Job(str):"Chef"			
		ID(uint):1	FN(str):"Frank"		LN(str):"Hill"
		ID(uint):2	FN(str):"Franz"		

1 5 9 17 25 33 41

(b) Neo4j storage architecture

Figure 3: Sample input data and the corresponding storage architecture in Neo4j for the given input data.

System Overview: During ingestion, the *Graphlet Manager* runs *cost-based graphlet chunking* (*CGC*) (§4.1.2) to partition PGM vertices and edges into columnar *graphlets*; these are persisted by the *Storage Manager* in *graph-native storage*. The manager also constructs two auxiliary structures, maintained by the *Index Manager*: (i) compressed-sparse-row (CSR) indexes for rapid neighborhood traversal and (ii) a *schema index* (SI) for locating graphlets whose attribute sets match a query. In *Index Manager*, CSR indexes and a schema index enable rapid traversal and lookup of relevant graphlets at query time. Meanwhile, the *Catalog Manager* stores statistics such as label metadata and histograms that the query optimizer can use to estimate query costs. The *Catalog Manager* records label cardinalities, histograms, and other statistics consumed by the optimizer's cost model. When a Cypher query arrives, the *Graph Query Optimizer*, built on Orca, applies graph-aware transformation rules, cost models, and the *Graphlet Early Merge* (*GEM*) heuristic (§4.3.2) to prune the plan search space and produce an efficient execution plan. Finally, the *Vectorized Graph Query Processor* executes the chosen plan with specialized, vectorized operators, including the *shared schema row format* (*SSRF*) (§4.2.2), to accommodate heterogeneous graphlet schemas efficiently. Figure 2b illustrates a typical data lifecycle in TurboLynx, including *bulk insertion* of new data, *query processing*, and *bulk updates* to existing graphlets.

***Step 1**: Bulk Insert.* New vertices and edges (e.g., `Person` or `City` labeled data) are ingested in batches. The *Graphlet Manager* creates columnar *graphlets* via *CGC*, grouping nodes and edges based on similarity. For example, four incoming vertices (`v0`–`v3`) are grouped into three distinct graphlets (gl_1–gl_3). It also constructs an SI and CSR indexes to support fast lookups and traversals.

***Step 2**: Query Optimization & Processing.* Upon receiving a Cypher query, the *Graph Query Optimizer* performs two preprocessing steps. **(2.1.1)** Leveraging the *Schema Index*, it identifies the graphlets that hold the required labels (e.g., gl_1 and gl_2 for `FN`, gl_{11} and gl_{12} for `area`). **(2.1.2)** It applies *Graphlet Early Merge* to curb the explosion of the plan search space. **(2.1.3)** Orca then produces an optimized plan featuring graph-aware operators (e.g., `AdjIdxJoin`) and, when advantageous, chooses different join orders for separate graphlets. For example, the optimizer chooses a plan with a `UnionAll` whose two children are symmetric `IdxNLJoin` sub-plans that differ only in the order of their joins. In the left branch, the engine first scans the unioned graphlet $gl' = gl_{11} \uplus gl_{12}$ to read `City` vertices, filters them with the predicate `area > 100`, follows the `livesIn` edge set gl_{21} via an `AdjIdxJoin` to reach candidate `Person` vertices, and finally probes their attributes through an `IdxSeek` on gl_1, applying the regular-expression $Filter_{\texttt{FN =~ 'Fran.*'}}$. The right branch performs the symmetric sequence: it scans gl_2 to obtain `Person` vertices, applies the name filter, traverses gl_{21} to the associated cities, and concludes with an `IdxSeek` on gl' to enforce the predicate `area > 100`. Merging the results of these complementary orders with the unioned graphlet allows the engine to exploit indexes on both the `Person` and `City` graphlets while keeping the plan search space small. **(2.2)** Finally, the query processor executes the physical plan: during binary joins, data from gl_1 and gl_2 remain in columnar format, whereas gl_{11} and gl_{12} are materialized in the *SSRF*, illustrating how multiple physical layouts coexist within a single query.

***Step 3**: Bulk Update.* When additional `City` and `Person` vertices (`v10`–`v13`) arrive after the initial graphlets are built, TurboLynx **(3.1)** examines each new tuple's schema and appends the tuple to the most compatible existing graphlet. **(3.2)** If no graphlet meets the schema requirements, a new graphlet is created. Although the illustration focuses on inserts, TurboLynx also supports deletions and updates (§4.1.5); each batch can therefore interleave all three operation types while preserving the columnar, graphlet-based storage layout.

4 TURBOLYNX SYSTEM DESIGN

4.1 Storage Design for Schemaless PGM Data

A central challenge in property-graph storage is that nodes and edges can carry arbitrary attribute sets. Row stores such as Neo4j and Memgraph embed a full schema in every record, but the per-tuple interpretation cost drags down query speed. TurboLynx resolves this by clustering records with similar attributes into *columnar graphlets*, retaining column-store performance while remaining schemaless, and by adding a lightweight schema index that lets attribute-only predicates jump directly to the relevant graphlets.

4.1.1 Challenges of Designing Columnar Storage. A naïve columnar design assigns one schema to each label, treating unlabeled elements as having a special NONE label. This approach produces a single, extremely wide table in which most columns are null, wasting space and slowing scans. At the opposite extreme, creating a separate table for every unique schema [69] explodes the number of tables, overwhelming both the optimizer and the execution engine.

4.1.2 Our Approach: Cost-based Graphlet Chunking. Our objective is to strike a balance between two extremes: proliferating too many schemas, each tailored to a distinct set of attributes, and a single, monolithic schema that lumps all attributes together. Coarser schemas reduce catalog size but increase null density significantly in the presence of non-uniform data, resulting in wasted space and slower scans. However, a unified, single schema remains practical for highly uniform data, where null overhead is small. The right granularity, therefore, hinges on clustering records by attribute similarity while weighing heterogeneity against the cost of nulls.

Definition 1. Graphlet, Vertexlet, and Edgelet. To formalize our clustering approach, we define a *graphlet*, the basic storage unit in TurboLynx. We partition a property graph G into *vertexlets* (for N) and *edgelets* (for E). A *vertexlet*, denoted vl, is a set of vertices partitioning N (i.e., $N = \bigcup_i vl_i$), and all vertices in a vertexlet share the same label set, thus accelerating queries that filter by label. Let $\mathcal{N}$ denote the total set of vertexlets. For any label R, we define $\mathcal{N}(R) \subseteq \mathcal{N}$ to be those vertexlets with label R. Edgelets are defined analogously, with $\mathcal{E}$ the full set. Vertexlets and edgelets together constitute the collection $\mathcal{H}$ of *graphlets*, written $\mathcal{H} = \mathcal{N} \cup \mathcal{E}$.

Our clustering technique is called *Cost-based Graphlet Chunking (CGC)*. For two graphlets $gl_i, gl_j \in \mathcal{H}$, the *cost-aware similarity* between the two graphlets is defined as:

$$casim(gl_i, gl_j, \mathcal{H}) = c(\mathcal{H}) - c(\mathcal{H}'). \tag{1}$$

where c (defined below) specifies the query cost function and $\mathcal{H}'$ is $(\mathcal{H} - \{gl_i, gl_j\}) \cup \{gl_i \oplus gl_j\}$ formed by replacing two graphlets with a newly merged one. Operator $\oplus$ implies a merge operation of two graphlets. Here, the schema of the newly merged one is the union of the schemas of gl_i and gl_j. The function c is defined as a linear combination of these three components.

- The number of graphlets $|\mathcal{H}|$.
- The number of null entries $\Sigma_{gl \in \mathcal{H}} \Gamma(gl)$ where Γ counts the number of null values.
- Total query processing overhead $\Sigma_{gl \in \mathcal{H}} \Psi(|gl|)$ where Ψ models the slowdown from underutilized vectorization, inspired by Boncz et al. [13], who showed performance degrades as the vector size decreases. Concretely, $\Psi(|gl|) = \frac{\kappa}{|gl|}$ for $|gl| < \kappa$ (and 0 otherwise), with $\kappa = 1024$ as the vector size used in TurboLynx.

The first two components aim to find a balance between the total number of schemas and the number of null entries. The third component reflects our intention to treat vectorizability as a major factor. Ideally, we want to form the set of schemas such that tuples in the tables are in sufficient quantity to populate the vectorization pipeline. Let us introduce three linear coefficients, C_{sch}, C_{null}, and C_{vec}, for three factors above, respectively. Using these three factors and coefficients, we can express $c(\mathcal{H})$ as:

$$c(\mathcal{H}) = C_{sch} \cdot |\mathcal{H}| + C_{null} \cdot \Sigma_{gl \in \mathcal{H}} \Gamma(gl) + C_{vec} \cdot \Sigma_{gl \in \mathcal{H}} \Psi(|gl|) \tag{2}$$

We can also expand $\mathcal{H}'$ in a similar fashion and rewrite $casim()$.

$$\begin{aligned} casim(gl_i, gl_j, \mathcal{H}) = \; & C_{sch} + C_{null} \cdot (\Gamma(gl_i) + \Gamma(gl_j) - \Gamma(gl_i \oplus gl_j)) \\ & + C_{vec} \cdot (\Psi(|gl_i|) + \Psi(|gl_j|) - \Psi(|gl_i| + |gl_j|)) \end{aligned} \tag{3}$$

Using this metric, we determine that it is beneficial to merge two graphlets when $casim(gl_i, gl_j, \mathcal{H}) > 0$.

In TurboLynx, *CGC* clustering for a label l runs in two passes. In the first, the system scans the data once to record each tuple's schema and count how many tuples share that schema, assigning every distinct schema to its own provisional graphlet. These graphlets

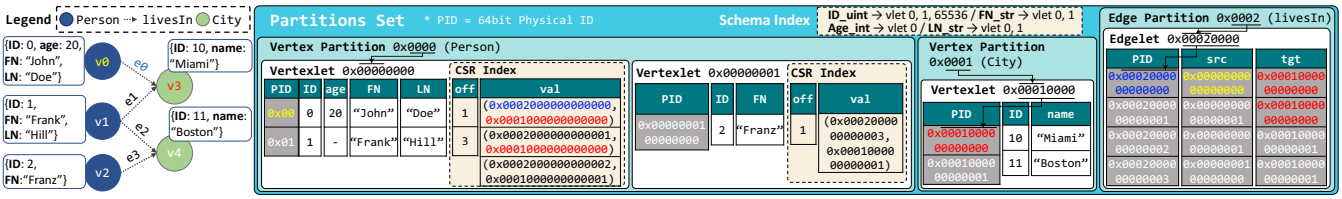


Figure 4: Example input graph and corresponding storage layout in TurboLynx's storage architecture. To clarify the mapping, we color-code v_0, e_0, and v_3 in both the input graph and the storage layout.

Algorithm 1: Cost-based Graphlet Chunking

Input: N^l (Nodes under label l)
Output: Corresponding *CGC* results *Res*

```
Function CostBasedGraphletChunking(N^l)
    H^l ← ExtractGraphlets(N^l);
    {H^l_i}_{i∈[1,p]} ← SplitIntoMultipleLayers(H^l);
    Res ← {};
    foreach i ∈ [1, p] do
        Res ← AgglomerativeClustering(Res ∪ H^l_i);
    return Res;
Function AgglomerativeClustering(H)
    while |H| > 1 do
        (gl, gl') = arg max_{gl_i, gl_j ∈ H} s(gl_i, gl_j);
        if s(gl, gl', H) < τ then break;
        H ← H − {gl, gl'} ∪ {gl ⊕ gl'};
    return H;
```

are then sorted by size and divided into p "layers," each containing graphlets whose tuple counts differ by no more than a threshold t. Starting with the largest layer, TurboLynx performs size-aware agglomerative clustering, repeatedly merging the most similar pair of graphlets, until the similarity score falls below a user-defined threshold τ. The procedure is repeated for the remaining layers in descending size order, which prevents *CGC* from prematurely fusing many tiny graphlets and degrading cluster quality. The second pass re-reads the dataset to materialize the final graphlet assignments. Both the layer width t and the similarity cutoff τ (default 0) are tunable parameters.

4.1.3 Schema Indexing for Query Processing Acceleration. Label-agnostic queries—those mentioning only attribute names without label—are ubiquitous in GDBMS workloads because users often lack detailed knowledge of a graph's schema. The naïve response is to scan every vertex and edge for the requested attribute, which is prohibitively expensive. In row stores that embed a schema inside each record, the absence of an index forces the engine to parse every tuple's schema block even if the tuple does not contain the attribute. In contrast, our *CGC* clustering lets us maintain a lightweight schema index that restricts the search to graphlets known to contain the attribute, eliminating full scans.

Schema Index We introduce a Schema Index (SI), an inverted index that tracks which graphlets contain each attribute. SI enables fast processing of attribute-only queries by quickly identifying the relevant graphlets. Each keyword in the SI represents an attribute name augmented with the concatenation of its type, and a *posting list* corresponds to a list of graphlet IDs that contain the attribute. For each entry, we also store the attribute's position within the graphlet to avoid recomputing it. SI is formally defined as:

$$SI(\mathcal{H}, a) = \{i \mid \exists gl \in \mathcal{H} \ s.t. \ a \in sch(gl) \wedge i = id(gl)\} \quad (4)$$

where $id(gl)$ is a function that returns the unique id of a given graphlet gl and $sch(gl)$ returns the set of gl's attributes. We use the prefix operation to find the posting list corresponding to a given attribute in SI. (Recall that a SI keyword is a concatenation of an attribute name and its type.) When accessing more than one attribute $(a_1, a_2, ..., a_n)$, we retrieve the IDs of graphlets that contain all these attributes using *set intersection* with $\bigcap_{i=1}^{n} SI(\mathcal{H}, a_i)$.

In addition to the SI, we maintain for each label a list of graphlet IDs, allowing quick identification of graphlets by label. For queries with both label and attribute predicates, we intersect the label-based IDs with the SI's attribute-based IDs to find the relevant graphlets.

4.1.4 Storage Architecture. Figure 4 shows our detailed storage design with sample data. A graph dataset is stored in a partition set that includes both vertex and edge partitions, each for a particular label set. Each partition is subdivided into one or more *vertexlets* or *edgelets*, where attributes are stored in a columnar format.

ID scheme. We assign each partition a 16-bit identifier, concatenate it with a 16-bit local offset to obtain a 32-bit *graphlet ID*, and then create a tuple's 64-bit *physical ID (PID)* by appending a sequence number. The PID lets us compute a tuple's file offset directly, eliminating the need for a separate index.

CSR (Compressed Sparse Row) index. Vertex connectivity is stored in CSR form as extra columns within each vertexlet. Every chunk contains an *offset* array that marks the start of each vertex's adjacency list and a *value* array holding ⟨⟨*neighbor vertex PID*, *edge PID*⟩⟩ pairs, allowing direct access to both neighboring vertices and their edges without maintaining a separate index.

4.1.5 Handling Updates. We support batch-style updates, wherein modifications are applied together without interleaved queries. The operations we consider are: 1) insert, 2) delete, and 3) update (= delete & insert). Inserts rely on the SI. Given a tuple with schema $\{a_1, a_2, ..., a_n\}$ and label l, we identify a set of graphlets with $\bigcap_{i=1}^{n} SI(\mathcal{H}^l, a_i)$. The tuple is then appended to the graphlet whose schema most closely matches the tuple's schema ($\arg\min_{gl} |sch(gl) - sch(tuple)|$, where *sch* returns the schema). If no existing graphlet meets the conditions, we create a new one with the same schema as the tuple and append the tuple to it. When a new graphlet gl' is created, for each attribute $a_i \in sch(gl')$, we insert $id(gl')$ into $SI(\mathcal{H}^l, a_i)$. To prevent an accumulation of numerous small graphlets, newly created ones are periodically checked, and a full *CGC* recomputation is triggered once the query processing overhead term ($\Sigma_{gl \in \mathcal{H}} \Psi(|gl|)$: the 3rd term in Equation 2) exceeds a configurable threshold (see §5.2.2). For deletions, each graphlet maintains a deletion vector marking removed tuples. During execution, tuples with the deletion bit set are ignored.

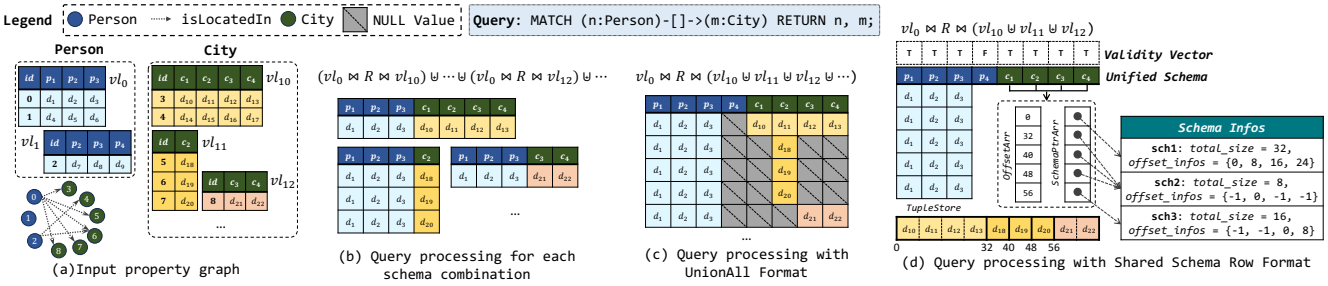


Figure 5: Challenges in the query processing of binary operators and TurboLynx's Shared Schema Row Format (SSRF) technique.

4.2 Schemaless Graph Query Processor

Our adoption of a *columnar* storage format influences the design of the query processing. While it brings us two key benefits, 'the ability to introduce an inverted index' and 'vectorization friendliness,' it raises a few other issues from the introduction of the graphlet concepts. First, query operators must now handle *multiple schemas* as input, because a single label in the query (e.g., `:City`) can map to multiple graphlets, each with its own schema. A more serious issue is the intermediate schema explosion in binary joins, since the operator needs to handle all combinations of graphlet pairs from two sets. This may worsen as the degree of multi-join increases.

4.2.1 Challenges.

(1) Processing Overhead from Multiple Schemas. Although *CGC* reduces the number of schemas per label at the storage level, the resulting graphlets can still be numerous during query execution. One major overhead is managing the expression tree. Each schema variant demands its own tree with adjusted column references. For binary operators, we must maintain mappings that specify how columns from each input map to the output schema, further increasing memory usage. As a result, query processing overhead often grows with the number of schemas involved. In our evaluations, a single label can have up to 1410 graphlets in a real-world dataset.

(2) Explosion of Intermediate Schemas: The Schema Bloating Problem. Beyond per-schema overhead, TurboLynx must address the exponential explosion of intermediate schemas in binary joins. The number of intermediate schemas for the join operation can be at most $n \times m$ schemas, where n and m are the graphlet sizes of the two join operands. Consider the simple multi-hop traversal query '`MATCH (a:A)-[]->(b:B)-[]->(c:C)`'. The number of possible schema combinations can grow exponentially, up to $|\mathcal{H}(A)| * |\mathcal{H}(B)| * |\mathcal{H}(C)|$ in the worst case. We refer to this issue as the *schema bloating problem.* Since multi-hop traversals are common in graph queries and are often emphasized as a key strength of GDBMS, it is crucial to address this issue in TurboLynx.

4.2.2 Our Approaches.

(1) Mitigating the Schema Processing Overheads via Unified Schema. To handle a large number of graphlet schemas efficiently, we introduce a *unified schema* with a *validity vector.* The unified schema is formed by the union of fields from all the schemas, while the validity vector indicates which fields are valid for the current schema under processing. Figure 5d depicts the concept. This design offers several advantages. First, it avoids wasted memory from columns that a given schema does not actually require, since the validity vector can skip them rather than storing null entries. Second, fixed field positions within a unified schema enable both a single expression tree and straightforward columnar processing: operators can execute without worrying about correct column positions in each graphlet's schema. In addition, the validity vector can accelerate operations such as filtering, aggregation, column copying, and hashing, since column nullity checks reduce to a single bit test.

(2) Shared Schema Row Format for the Schema Bloating Problem. While a unified schema reduces per-schema overhead, it is insufficient for binary joins. Figure 5b shows the schema combinations generated from one `Person` graphlet vl_0 and all `City`'s graphlets (the set using vl_1 is omitted for conciseness). For *multi-hop* joins, maintaining all schema combinations quickly becomes excessive. Moreover, applying the unified schema in these joins is also insufficient, because columns can remain mostly null (e.g., c_1, c_3, c_4 in Figure 5c). The validity vector is of little help in reducing null entries here since these *sparse columns* are still considered valid.

To address the issue, we introduce a *Shared Schema Row Format* (*SSRF*). *SSRF* adopts a partial row format representation for the data in the columns that belong to either operand (Figure 5d). Only the columns c_1-c_4 are stored in the row format, eliminating nulls from sparse columns in the intermediate data. Since tuples in row format may differ in schema, we keep schema definitions in a separate 'schema infos' table (Figure 5d), allowing all tuples with the same schema to share one entry. This method mitigates the schema-bloating problem because it avoids materializing an exhaustive set of schema combinations. Instead, only the schema lists of one operand are maintained separately. The *SSRF* technique that combines these ideas follows our principle of *separating the schema information and tuples*, rather than embedding the schema in each tuple.

Figure 5d shows how these concepts are put together into action. The `OffsetArr` stores the byte offsets of each tuple in `TupleStore`, and each tuple points to its schema information. The information stores *i*) the size in bytes of the tuple belonging to the corresponding schema (`total_size`) and *ii*) `offset_infos` that indicates the offset of the attributes in each tuple. By combining `OffsetArr` and `offset_infos`, we can access each tuple's attribute. A value of -1 in `offset_infos[i]` indicates that the i-th attribute is null.

4.3 Schemaless Graph Query Optimizer

Our strategy in building TurboLynx's query optimizer is to apply as few modifications as possible to a mature open RDBMS query optimizer, Orca [60]. Existing query optimizers developed primarily for RDBMS need to be adapted on multiple fronts to be integrated into the schemaless graph query processing pipeline of TurboLynx.

4.3.1 Challenges.

There are two main challenges in building a query optimizer for the schemaless PGM graph data.

(1) Mismatch of RDBMS Query Optimizers for the Schemaless Graph Query. RDBMS optimizers assume that all tuples in a table

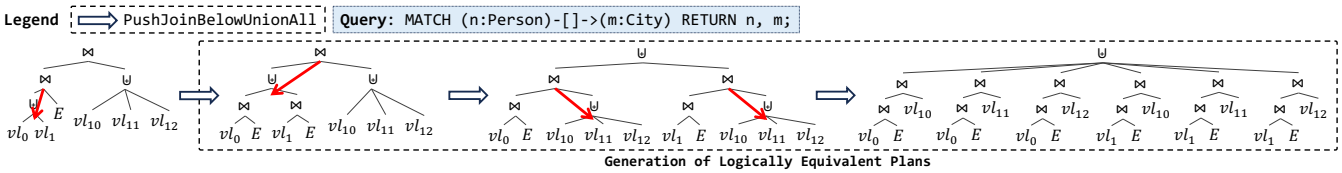


Figure 6: Identifying logically equivalent plans for a given query through the `PushJoinBelowUnionAll` rule. Red arrows indicate which joins are pushed below the `UnionAll`(⊎). We omit the enumeration of all possible plans.

follow the same schema, whereas vertices or edges in PGM may not. Further, unlike SQL queries that specify the tables explicitly, graph queries allow users to omit labels. This necessitates graph query optimizers to identify target nodes or edges (and graphlets for TurboLynx) from the limited information (e.g., accessed attributes) available within the query. Lastly, many RDBMS operators, rules, and cost models do not directly translate to graph query optimization.

(2) Plan Search Space Bloating Problem. The plan search space for the PGM data can become intractably large if many graphlets are involved. Each node pattern in a Cypher query is translated into scans of multiple graphlets combined via `UnionAll` (⊎), while each edge pattern is translated into two joins (source-edge and edge-target). During Orca's exploration phase, the `PushJoinBelowUnionAll` rule converts a 'join of `UnionAll`' into a '`UnionAll` of joins' (red arrows in Figure 6), distributing the join across each branch of the `UnionAll` and increasing the number of joins (the right-most plan in Figure 6). Consecutive joins at multiple levels are then further transformed by the associativity and commutativity rules, leading to an explosion of logically equivalent plans and excessive compilation time. Disabling the rule forces a single join order across all graphlets, eliminating opportunities for more efficient plans.

4.3.2 Our Approaches.

(1) Adapting Operators, Rules, and Cost Model for the Graph Query. We extend Orca for TurboLynx by adding graph-aware operators, transformation rules, cardinality estimators, and cost functions. Simple operators map onto existing relational counterparts (e.g., `Scan` → `UnionAll+TableScan`, `AdjIdxJoin` → `IndexNLJoin`). For graph-specific operators, we introduce six logical operators (`VarlenJoin` and `ShortestPath`, etc.) that encapsulate traversal metadata (e.g., hop bounds, labels) and four physical operators (`PhysicalVarlenAdjIdxJoin`, `PhysicalShortestPath`, etc.). Logical operators are implemented by adapting Orca's four existing logical operators (e.g., `IndexApply`). We introduce two logical transformation rules, extending Orca's XformJoin2IndexApply, so that the optimizer can generate index-based plans for variable-length traversals and graphlet scans. We also add four physical transformation rules that lower these logical operators to their physical counterparts. The traversal metadata is preserved through these mappings and utilized by the cost model.

Given the difficulty of accurate cardinality estimation for path expression queries as well as shortest path queries (i.e., recursive queries in relational databases), an area where research has yielded limited success, systems often employ heuristics. For instance, PostgreSQL assumes 10 recursive steps and scales recursive term cardinalities by this factor. Similarly, we adopt the methodology used in Kuzu implementation, which multiplies the estimated one-hop cardinality by the maximum hop count to approximate the overall cardinality for path expressions. Despite this coarse heuristic, TurboLynx still outperforms other engines on these queries thanks to its efficient columnar graphlet storage and graph-aware operators.

We base our cost model on Orca's Greenplum formulation [45] but refine relevant operators, including the scan, so that its I/O cost is computed per graphlet, using each graphlet's actual average tuple width instead of a single global row width. This adjustment captures the heterogeneity of the schema and yields more accurate estimates. We further adjust the `IndexNLJoin` cost model to omit random-I/O penalties when probing CSR indexes, because these indexes are kept entirely in memory and incur no disk seeks.

(2) Managing Plan Search Space with Graphlet Early Merge (GEM). To mitigate the plan search space explosion from a large number of graphlets, we introduce an early merge of graphlets to reduce the number of input tables to the optimizer. This phase groups and merges graphlets into coarse-granular *virtual graphlets* in a way that graphlets with similar join orders are grouped during the query optimization. A virtual graphlet is purely conceptual, merging multiple graphlets so the optimizer sees fewer input tables without altering the physical data. For example, merging all graphlets under a `UnionAll` operator into one virtual graphlet forces a uniform join order across all graphlets. By contrast, partitioning graphlets into multiple groups allows different join orders to be explored for each group, providing more optimization opportunities. Increasing the number of groups further expands the plan search space, potentially yielding better plans but at the cost of additional time.

Determining the optimal grouping of graphlets is a major challenge. We introduce heuristics with a time-out to make the search manageable. The sketch of *GEM* is as follows.

(1) Form random groups of graphlets, with each group merged into a virtual graphlet (two groups by default).
(2) Enumerate logically equivalent plans using the `PushJoinBelowUnionAll` rule and selectively evaluate a subset of these plans.
(3) Determine the optimal join order for each alternative plan using Greedy Operator Ordering [25] and then compute costs.

We repeat this process within a predefined time limit for each node pattern present in the input logical plan. As a result, the lowest-cost plan is passed to Orca, which further optimizes it without the `PushJoinBelowUnionAll` rule. This enables us to explore a broader search space at a manageable computational cost.

4.4 Implementation

We implemented a prototype of TurboLynx by partially reusing existing codebases (including DuckDB [53] and Orca), totaling about 246k LOC. Specifically, we adopted DuckDB's expression evaluator, built-in functions, and major data structures (e.g., Vector and DataChunk). These components are already well-optimized and thoroughly tested, so reimplementing them would require considerable effort with little benefit to our contributions. On top of this foundation, we instead added about 57k LOC that implement a native graph-centric storage layer, extend operators for schemaless queries, and retrofit Orca with graph-aware rules and cost models.

Although Kuzu likewise reuses some core functions (e.g., arithmetic and string operations) and parts of its type system (e.g., date and timestamp) from DuckDB, TurboLynx significantly outperforms it by up to 106.89×, showing that DuckDB's read-optimized internals alone cannot solve the core challenges of graph databases: heterogeneous schemas, graph-specific operators, graph query optimization, and plan space bloating problems.

5 EVALUATION

We evaluated TurboLynx in the following aspects.

- Sensitivity analysis of TurboLynx's components: The effect of *CGC* (§5.2.1), *SSRF* (§5.2.3), and *GEM* (§5.2.4).
- End-to-end analytic query performance on the *schemaful* data (§5.3.1, §5.3.2): We evaluated end-to-end query time on LDBC and TPC-H. TurboLynx showed up to 183.9× better performance on the LDBC and 44.63× on the TPC-H against competitors.
- End-to-end query performance on *schemaless* graph data (§5.3.3): On DBpedia, TurboLynx showed up to 86.14× better performance against competitors.

5.1 Experimental Setup

5.1.1 Workloads. We used the following three workloads in our evaluations. For LDBC SNB Interactive and TPC-H, we applied the scale factors of 1, 10, and 100.

- Social network benchmark LDBC SNB Interactive [23]: We evaluate all fourteen complex queries. The complex queries execute multi-step graph pattern matches that span several entities and relationships.
- Decision support benchmark TPC-H: It is a well-known decision-making OLAP benchmark. To evaluate it in GDBMSs, we converted each table to vertices or edges according to whether it represents an entity or a relationship. Entity tables became vertices, while PARTSUPP, as a relationship table between PART and SUPPLIER, was converted to edges with its attributes. We also converted PK-FK relationships into edges. All SQL queries were translated into Cypher via SqlTranslator from Neo4j.
- DBpedia: A real-world knowledge graph with highly diverse schemas (2796 unique attributes, 282764 unique attribute sets). We used the 2016-04 Core release of DBpedia, whose compressed size is about 48 GB (∼9.5 billion RDF triples). Following Sun et al. [62], we converted DBpedia into the PGM format. The resulting graph has unlabeled nodes and edges with exactly one label. For RDBs, unlabeled nodes were loaded into a single large table, while each edge label was loaded into its corresponding table. We then used the 20 queries introduced in [62], converting them from SPARQL to Cypher via existing works [2, 73].

Table 1 summarizes the dataset statistics (e.g., vertex and edge counts). Although LDBC and TPC-H have predefined schemas, we include them to demonstrate that TurboLynx can handle both fixed-schema and schemaless workloads effectively.

5.1.2 Competitors. We compared TurboLynx to five GDBMSs (GraphScope, Memgraph, Neo4j, Kuzu, DuckPGQ) and two RDBMSs (Umbra, DuckDB). GraphScope [24, 59] and Memgraph are analytical graph engines, while Neo4j is a widely used, stable GDBMS baseline. Kuzu [26] and DuckDB are read-optimized systems targeting analytical workloads, while Umbra is an HTAP DBMS that also provides strong support for analytical queries. We also evaluated

Table 1: Dataset statistics. "Rel." and "PGM" are the sizes of the preprocessed relational and PGM representations.

Dataset	SF	$\|V\|$	$\|E\|$	Rel. (GB)	PGM (GB)
LDBC SNB	1	3.2M	17.3M	0.7	1.3
	10	30.0M	177M	7.2	14
	100	283M	1.78B	75	140
TPC-H	1	8.66M	–	1.1	1.7
	10	86.6M	–	11	14
	100	866M	–	106	141
DBpedia	–	77M	227M	213	19

DuckPGQ, which is a DuckDB extension for SQL/PGQ property graph queries. Although some systems have enterprise or advanced editions, we use publicly available community versions in line with common research practice. Thus, our results do not imply an absolute ranking but rather show how TurboLynx compares to these representative baselines in analytical scenarios. See Table 2 for details.

5.1.3 Execution Environment. We used a machine with two Intel Xeon Gold 6130 @2.1 GHz CPUs and 512 GB RAM, running Ubuntu 20.04 (Linux 5.4.0-205-generic). TurboLynx was compiled with GCC 17 and G++ 17. We used OpenJDK 15.0.10 for Neo4j. All systems were executed in a single-threaded mode for fair comparison. For memory setting, we configured TurboLynx, Memgraph, GraphScope, Kuzu, Umbra, DuckDB, and DuckPGQ each to use up to 500 GB of memory. Neo4j was configured according to its official guidelines, with the initial and maximum Java heap size set to 30 GB, a page cache allocation of 470 GB.

5.1.4 Measure. For each query, we measured the geometric mean of five end-to-end query execution times after three warm-up runs. We set a timeout of 1 hour and reported the query execution time of 1 hour for timeouts. Queries that failed were excluded from the comparison. In the end-to-end query performance graphs, we mark T (time-out), F (fail to load), or X (other reasons) for failed queries.

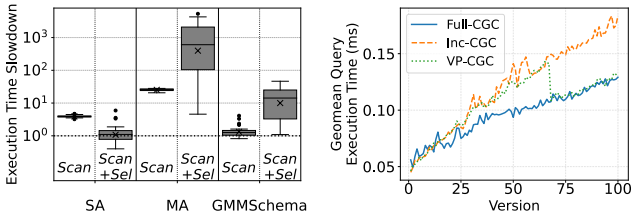
5.2 Sensitivity Analysis

We conducted various sensitivity analyses of TurboLynx's performance using the DBpedia dataset. The goal of these experiments was to assess the impact of the techniques applied to support the schemaless workload in TurboLynx.

5.2.1 The efficacy of CGC. To evaluate *CGC*, we compared it with two extremes from §4.1.1 – *i)* merging all graphlets into one large graphlet (**MA**), and *ii)* creating a distinct graphlet for every unique attribute set (**SA**, for "separating all"). We also evaluated GMM-Schema [15], a state-of-the-art schema discovery method for property graphs. For each approach, we clustered the DBpedia data accordingly and loaded each cluster into TurboLynx as a graphlet. *CGC* involves tunable weights C_{sch}, C_{null}, and C_{vec} that control its clustering behavior (See Eq. 2). Intuitively, the larger C_{sch} favors merging graphlets with dissimilar schemas. The larger C_{null} penalizes layouts with many NULLS (favoring denser layouts when memory or scan bandwidth is tight). Increasing C_{vec} encourages forming larger graphlets that are more amenable to vectorized scans in OLAP-style workloads. On DBpedia, setting C_{null} too high could result in up to 282K graphlets, many of which were small, impacting vectorization performance and increasing the optimizer's overhead. Thus, we use a single reasonable default weight set empirically chosen to work well across our experiments, $C_{\text{sch}} = 100$, $C_{\text{null}} = 0.3$, and $C_{\text{vec}} = 10000$. We then ran 30 randomly generated queries, focusing on core storage operations (scan and scan with selection), and measured compilation and execution times for each method.

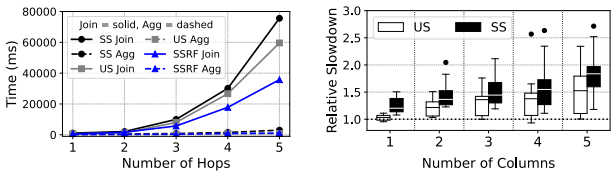
Table 2: Details of competitor database systems used in the evaluations.

	Data Model	Storage Format	Data Scheme	Supported Language	Evaluated Version	Note
Neo4j	Property Graph	Row-based	Schema-free	Cypher	v2025.03.0 Community Edition	• followed the same settings as the reference impl‡ (unique constraints on the `id` or PK attribute(s)), while omitting property indexes to maintain fairness across systems • used the 'aligned' store format, which is the default
Memgraph	Property Graph	Row-based	Schema-free	Cypher	v3.0	• used identical setting as Neo4j • used IN_MEMORY_ANALYTICAL mode
GraphScope	Property Graph	Columnar	Single Schema	Cypher	v0.31.0	• evaluated with Graph Interactive Engine (GIE) of GraphScope • used queries written in Cypher instead of C++ (stored procedure) • excluded LDBC C1, C5-C6, C10, and C12-C14 queries since GIE only partially supports Cypher • excluded all TPC-H queries since GIE crashes on most of them
Kuzu	Property Graph	Columnar	Pre-defined Single Schema	Cypher	v0.8.2	• added PK constraints on the `id` or PK attribute(s) • specified the multiplicity (e.g. many-to-one) for each edge
DuckPGQ	Property Graph	Columnar	Pre-defined Single Schema	SQL/PGQ	947eb8d	• used the same experimental settings as DuckDB • queries not yet supported in DuckPGQ's SQL/PGQ (e.g., optional match) were executed in SQL
Umbra	Relational	Columnar	Pre-defined Single Schema	SQL	v25.07.1	• followed the same settings as the reference impl‡ (PK constraints on the `id` or PK attribute(s), and PK-FK constraints)
DuckDB	Relational	Columnar	Pre-defined Single Schema	SQL	v1.2.0	• followed the same settings as the reference impl‡ (PK constraints on the `id` or PK attribute(s), and PK-FK constraints)



(a) Efficacy of CGC. (b) CGC under update.

Figure 7: CGC performance effects and periodic update.



(a) Varying Num Hops. (b) Varying Num Columns.

Figure 8: Efficacy of SSRF.

Figure 7a shows normalized execution times relative to *CGC*. The scan with selection achieves up to 5319× improvement, and scan sees up to 28× speedup from reduced null overhead. MA produces about 212 billion null entries for 77 million nodes, while SA leaves many tiny graphlets unmerged, causing overhead (§4.1.2). GMMSchema selects the n most frequent properties (n=1 by default) as a "base schema" and computes similarity among schemas by counting how many properties lie outside it. Hence, even unrelated schemas with a similar *number* of properties that are not in the "base schema" may be merged, leading to suboptimal performance. In terms of compilation time, MA (a single schema) and GMMSchema (fewer overall schemas) are only 15% and 6% faster than *CGC*, whereas SA's large number of schemas yields a 2.22× slower compile time.

5.2.2 Update. We evaluate *CGC* under an update workload on `DBpedia` that accumulates many small graphlets. Starting from a 10K-tuple snapshot, we apply 100 updates of 1K tuples; each update contains about 200 distinct attribute sets (schemas), all sampled from `DBpedia`. We compare three policies: Inc-CGC (incremental placement with no merges), Full-CGC (full recomputation after every update), and VP-CGC (vectorization-penalty triggered: run a full *CGC* when the penalty exceeds a fixed threshold θ; we set $\theta = 200$ empirically). For each update version, we measure query execution time using the same queries used in §5.2.1.

Figure 7b shows that average query time increases as updates accumulate, with Inc-CGC steadily slowing relative to Full-CGC. The main reason is the proliferation of small graphlets, which produces under-filled vectors and reduces vector utilization. By contrast, VP-CGC runs a full *CGC* once the penalty exceeds θ. After a merge, subsequent updates mostly append to existing graphlets rather than creating new small ones, so the penalty grows slowly and performance remains close to Full-CGC. If a stream of genuinely new attribute sets arrives, small graphlets accumulate again; once the penalty crosses θ, another merge is triggered, restoring the layout. For VP-CGC, the average update time is 5.1s and the merge takes 10.4s.

5.2.3 The efficacy of SSRF. To present how much impact the *SSRF* technique has, we measured the speed-up gained by employing *SSRF* compared to *i)* using a straightforward approach of a full combination of intermediate schemas (Figure 5(b)) and *ii)* using a simple unified schema (Figure 5(c)). They are denoted as **SS** and **US** in Figure 8, respectively. We evaluated *SSRF* in two scenarios.

(i) Increasing hop count in graph traversals. First, we demonstrate that *SSRF* reduces null overhead and mitigates schema bloating in graph traversal queries without forfeiting columnar benefits. To this end, we generated random traversal queries on `DBpedia` with hop counts ranging from 1 to 5, followed by additional aggregation steps. As shown in Figure 8a, *SSRF*'s traversal cost remains up to 2.1× lower than the baselines as the number of hops increases. Moreover, it incurs no notable penalty on subsequent aggregations.

(ii) Varying the number of returned attributes. Next, we examined whether *SSRF*'s performance gains grow more pronounced as queries return more attributes—i.e., if reduced null overhead becomes more advantageous when additional attributes are involved. To evaluate this, we selected `DBpedia` queries that involve traversal and varied each to return between 1 and 5 attributes (chosen at random). We then measured execution times for these queries. As shown in Figure 8b, US accumulates more null entries when returning many attributes, slowing subsequent processing. SS also shows suboptimal performance due to the overhead of processing all schema combinations. By contrast, *SSRF* avoids storing these large, sparse columns entirely, achieving up to a 2.6× speedup.

5.2.4 The efficacy of GEM. To evaluate the effectiveness of *GEM*, we derived several scenario queries based on `DBpedia` queries (Q7, Q10, and Q13). We measured query compilation and execution

‡https://github.com/ldbc/ldbc_snb_interactive_v1_impls

Table 3: Efficacy of GEM.

Time (ms)	Compilation	Execution	Sum	Speed-up
w/o GEM	845.7 ms	2596.9 ms	3442.6 ms	1.0
w/ GEM	898.5 ms	1904.3 ms	2802.8 ms	1.23

times for these queries with and without *GEM*. The output of *GEM* may favor a greater number of virtual graphlets among candidate plans which would increase the compilation time. However, we expected that the gain from the execution stage would be enough not only to compensate for this loss but also to deliver better overall performance. Table 3 shows that while the compilation overhead increased by 6.2%, the query execution time was reduced by about 26.7%, demonstrating the efficacy of *GEM*.

DBpedia data is suitable to demonstrate the utility of *GEM* because it is an evolving knowledge graph whose schemas vary greatly for the same entity type (e.g., persons, companies). These variations stem from factors like time, popularity, and domain-specific trends, leading to shifts in the distribution of connected nodes. For instance, in the domain of video games, schema differences are notable based on popularity. As a result, when the data is partitioned based on schema, different join orders will likely yield better query performance for each graphlet.

5.3 End-to-End Query Performance

We evaluated end-to-end query execution latencies on LDBC SNB Interactive, TPC-H, and DBpedia. The 'end-to-end' includes both the query compilation time and the actual execution of the plans. Table 4 summarizes the average relative performance of TurboLynx compared to its competitors across all evaluated benchmarks.

5.3.1 LDBC SNB Interactive. Overall, TurboLynx outperformed the best competitor, Umbra, by 2.53× and 7.74× at SF10 and SF100, respectively (Table 4). Figure 9 shows per-query end-to-end execution times. Among the queries, TurboLynx showed the largest speed-up of 4142× for the C8 query at SF100. The largest performance gap was observed with Kuzu, primarily due to join order selection, which introduced about 1.82 million times as many intermediate results. On the other hand, the performance superiority of TurboLynx was not as dramatic for C11. Because C11 applies filters to most of its graph patterns, even poor join orders yield small intermediate results, making performance less sensitive to plan quality.

Neo4j was slower than TurboLynx in all queries by 2.27× to 135.8×, particularly struggling with C3, which generated about 15.6M intermediate tuples requiring attribute accesses for a subsequent filter—nearly 181× more than other queries on average—and accounting for roughly 84% of the query's total cost. We observed that this performance difference was primarily due to the inefficiency of Neo4j's volcano query processing model and high storage access costs. Memgraph was also slower than TurboLynx for similar reasons as Neo4j but slightly outperformed Neo4j thanks to its in-memory optimized storage. However, it performed worse than Neo4j on some queries, generating inefficient plans that incurred more storage accesses (e.g., in C9, it required 424K× more accesses).

GraphScope shows poor results compared to other systems in queries like C3, which involved a large number of tuples to process. In contrast, GraphScope's performance was similar to or better than Neo4j for queries C2 and C11, which involved few tuples to process. We note that GraphScope supported only half of the queries, which hinders clear conclusions about its overall efficacy.

Kuzu showed lower performance, even though it targets graph query acceleration via worst-case-optimal joins (WCOJs) and factorized execution. It performed well for simple graph pattern matching but struggled with complex patterns due to suboptimal query optimization. For example, C3 and C6 can benefit from early filtering of 1–2 nodes, but Kuzu postponed filters until after costly pattern matching, resulting in execution times 1683× and 403× slower than TurboLynx (Figure 9 ①). These results underscore the central role of query optimization and the practical difficulty of matching the maturity of established optimizers. At the same time, the gap is not inherent: systems like Kuzu could narrow it by leveraging mature relational-optimizer techniques, rather than building an optimizer entirely from scratch. Note that Kuzu was unable to process C1 and C7 due to current limitations in its Cypher language support.

DuckDB exhibited the second-worst performance among competitors, primarily due to extensive use of hash joins for graph traversal [1], resulting in inefficient plans that scanned large data volumes. For example, in query C7, DuckDB scanned the entire likes edge table with 340 million tuples for building a hash table, whereas TurboLynx scanned only 60 tuples from adjacency lists. This was because the number of connected edges to traverse was small since the query started from a specific person node. Additionally, DuckDB did not work well for queries that required graph-specific operations (e.g., C12 and C13 (Figure 9 ②)).

Among all competitors, Umbra demonstrated the best overall performance. However, it was on average 7.74× slower than TurboLynx under SF100. This gap exists despite Umbra's data-centric code generation and its versatile join implementation, including WCOJs, designed to push relational performance to the limit. The difference is most pronounced in C13 and C14, where the shortest-path computation benefits from TurboLynx's specialized graph operators.

5.3.2 TPC-H. TPC-H queries typically scan large data volumes with only a subset of attributes and perform complex analytical operations (e.g., group-by, aggregation), making them amenable to vectorized processing and columnar storage. The benchmark can highlight the strength of TurboLynx's columnar design over the row-based storage. We are also interested in evaluating the efficacy of query optimizers in the GDBMSs, as the impact of suboptimal plans is more pronounced in long-running analytical queries.

Figure 10 presents individual query execution times. Overall, TurboLynx outperformed three GDBMSs by *an order of magnitude* (see Table 4), except for Kuzu at SF1 (8.37×). Compared to two RDBMSs, TurboLynx exhibited competitive performance (0.58× against Umbra, 1.53× against DuckDB at SF100). The largest performance difference was a 2528× slowdown on Kuzu's Q19 at SF10.

Neo4j and Memgraph suffered significant slowdowns due to row-based storage with embedded schemas. For instance, Q1 scans LINEITEM (7 of 16 attributes) followed by filter and aggregation, yet both systems incur substantial overhead due to runtime schema interpretation (Figure 10 ①). To pinpoint the cause, we profiled four representative queries (Q1, Q16, Q18, Q19) [42]. Since only Memgraph offers operator-level profiling, we analyzed it in depth. Memgraph's filter and group-by/aggregation operators were 132.6× and 27.1× slower than TurboLynx. Notably, these operators account for 34–98% of each query's total runtime. These results show that schemaless GDBMSs incur significant overhead from embedded schemas and inefficiencies in complex filter and aggregation.

Table 4: End-to-end benchmark results as TurboLynx's average speed-up over competitors. Each entry is the geometric-mean execution time (ms), shown as *competitor* / TurboLynx (unit 'ms' omitted). Within each scale factor, denominators vary because TurboLynx is averaged over only the queries each competitor successfully ran to make a fair comparison.

	Benchmark→	LDBC SNB Interactive						TPC-H						DBpedia	
	Scale Factor→	SF1		SF10		SF100		SF1		SF10		SF100		-	
GDBMS	Neo4j	6.46×	248 / 38	9.03×	404 / 45	10.47×	880 / 84	14.33×	2473 / 173	20.07×	20295 / 1011	15.73×	192699 / 12253	86.14×	41596 / 483
	Memgraph	3.20×	114 / 36	5.72×	256 / 45	11.74×	967 / 82	20.34×	3281 / 161	44.63×	39591 / 887	-×	-	-×	-
	Kuzu	12.27×	475 / 39	45.90×	1988 / 43	106.89×	8394 / 79	8.37×	1477 / 176	17.13×	17187 / 1003	14.34×	175100 / 12205	18.88×	8634 / 483
	GraphScope	5.78×	213 / 37	12.70×	451 / 36	26.92×	1451 / 54	-×	-	-×	-	-×	-	-×	-
	DuckPGQ	4.71×	180 / 38	28.83×	1289 / 45	183.9×	15905 / 84	1.02×	178 / 173	1.62×	1639 / 1011	1.54×	18924 / 12253	20.23×	9771 / 483
RDBMS	Umbra	0.93×	107 / 38	2.53×	684 / 45	7.74×	2319 / 84	0.43×	915 / 173	0.45×	10386 / 1011	0.58×	113512 / 12253	23.07×	11139 / 483
	DuckDB	2.87×	110 / 38	14.59×	652 / 45	41.27×	3467 / 84	1.02×	177 / 173	1.63×	1652 / 1011	1.53×	18801 / 12253	20.22×	9764 / 483
	Average	4.07×		11.81×		29.78×		3.21×		5.13×		3.15×		27.37×	

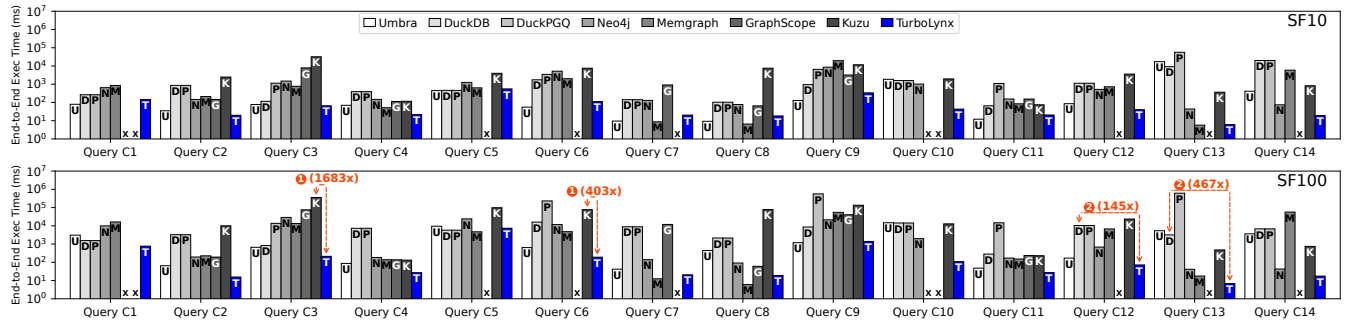


Figure 9: LDBC SNB Interactive SF10, SF100 Results. (SF1 results omitted due to space constraints.)

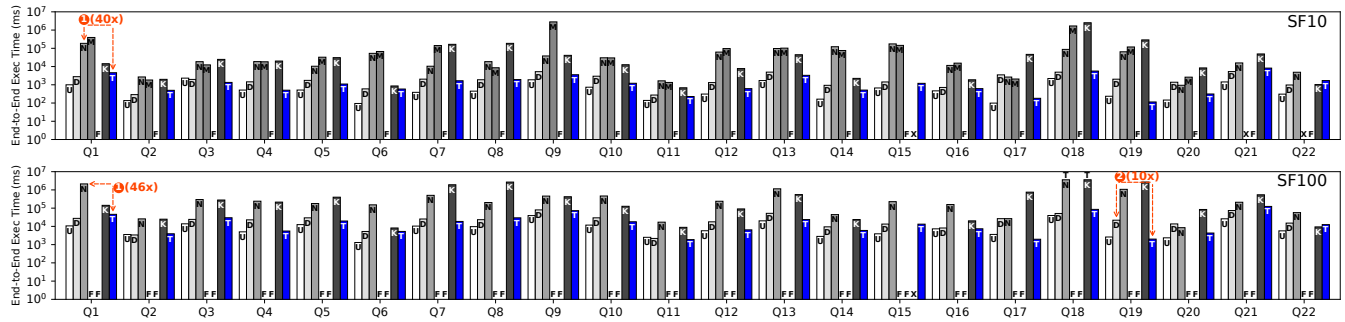


Figure 10: TPC-H SF10, SF100 Results. (SF1 results omitted.)

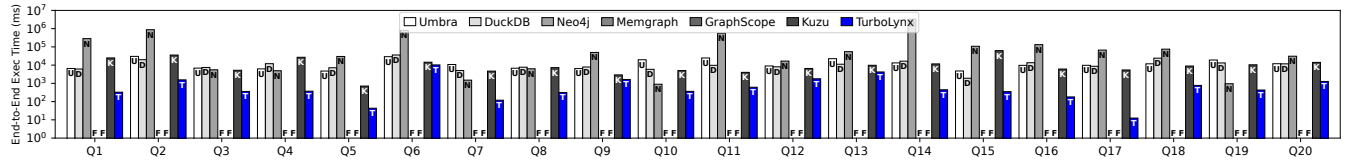


Figure 11: End-to-end query execution time comparisons on DBpedia data.

Kuzu uses a columnar format with predefined schemas, often outperforming row-based designs. However, certain queries (e.g., Q5 and Q7) experienced performance issues associated with join order selection, while another query (Q19) showed inefficiencies related to filter pushdown, leading to more expensive join operations between `PART` and `LINEITEM`. Additionally, Q15 was not supported due to limitations in its current coverage for Cypher language.

Umbra exhibited the best performance among competitors. It employs data-centric code generation that fuses complex expressions and operators into compact, low-latency execution pipelines. This approach has been shown in prior studies [42, 51] to outperform vectorized execution in many analytical workloads, enabling Umbra to reach near hand-optimized performance. DuckDB, while highly optimized for analytical workloads, showed the second-best performance among competitors. Nonetheless, TurboLynx outperformed DuckDB in most TPC-H queries (15 out of 22, at SF100). This advantage arose from adjacency-based lookups (e.g., Q19) where TurboLynx avoided scanning much of `LINEITEM`, lowering the scan volume by a factor of 105× (Figure 10 ②).

5.3.3 DBpedia. The queries in the `DBpedia` workload were generated by analyzing logs from the official `DBpedia` query endpoint, selecting frequently executed queries, and clustering them based on query similarity [48]. Thus, the queries represented typical usage patterns for graphs with a highly diverse schema. Through evaluations with the `DBpedia` workload, we seek to verify how well TurboLynx can handle not only schemaful workloads but also schemaless PGM workloads. Note that we excluded Memgraph and GraphScope because they failed during data loading. For Umbra, the same

single-table loading as with DuckDB led to a crash during loading. Thus, we loaded only the 'id' column (shared by all vertices) as a dedicated attribute and stored the remaining attributes as JSONB.

Our analysis highlighted two key factors. First, without labels on nodes, leveraging edge labels becomes critical. The best strategy is to scan edges with low cardinality and traverse the connected nodes from them. Second, queries that filter on node attributes must locate specific columns, which favors our graphlet-based storage.

Overall, TurboLynx outperformed the best competitor, Kuzu, by 18.88× (Table 4). Figure 11 provides end-to-end individual query execution time on DBpedia. TurboLynx achieved the largest speed-up of 7377× on Neo4j's Q14, which filters nodes by specific attributes. TurboLynx efficiently processes such queries by leveraging the SI to identify graphlets containing the target attributes quickly. On the other hand, TurboLynx showed the least performance improvement (1.40×) on Q6 because it is a simple one-hop traversal query that all other competitors can handle reasonably.

Neo4j showed the worst performance overall. Neo4j generated reasonable plans by leveraging edge label information to traverse the graph. However, in most cases, its performance was significantly degraded by high storage access costs. This issue was particularly apparent in DBpedia, which contained many attributes as Neo4j could not locate the attributes efficiently (e.g., Q14, Q16, Q17 etc.).

6 RELATED WORK

For storing graph data, modern GDBMSs employ diverse storage backends. Some adopt key-value stores [21, 35, 58], document stores [7, 30, 54], or wide-column stores [19, 33, 39] as their backends. They benefit from well-tested, mature NoSQL databases but suffer in complex graph queries because these storage systems and optimizers—initially designed for generic NoSQL use—lack graph-specific structures and operators. In contrast, TurboLynx's storage is designed to be a graph-native storage with graphlet concepts and *CGC* optimization along with novel indexing schemes.

Several GDBMSs have been developed based on RDBMS [29, 61, 62, 65], aiming to maximize the utilization of RDBMS techniques developed over decades. IBM DB2 Graph [65] receives a Gremlin query, converts it to SQL, and performs the query within DB2. SQL-Graph [62] stores graph data using the JSON capabilities provided by RDBMS to store vertex properties and adjacency information. However, they rely on relational operators and storage, providing neither native graph features (e.g., CSR-based indexing) nor addressing the null-overhead problem arising from the PGM. In comparison, the storage and query processor of TurboLynx are designed for the PGM. Although we utilize Orca [60], we modified it to be graph-aware, optimizing it to handle graph queries effectively.

Several recent studies aimed at better supporting graph queries by directly modifying the internals of RDBMS [32, 40, 64]. GRainDB [40] extends DuckDB by integrating the adjacency index and implementing various join operators optimized for graph data. DuckPGQ [64] extends DuckDB to facilitate querying property graphs. However, none of these systems accounts for PGM's schemaless nature. In contrast, TurboLynx supports the schemaless PGM via *CGC* and an SI, enabling fast querying over schemaless data.

Several techniques have been proposed for efficiently storing and querying semi-structured data [22, 63, 69]. JSON Tiles [22] organizes raw JSON into an optimized binary format and materializes frequent keys as columnar tiles to accelerate analytical queries. For log data, μSlope groups identical-schema tuples into per-schema tables and maintains a *schema map* of schema definitions. At query time, it scans this map to identify schemas that match the KQL predicates, quickly pruning the subsequent log search. This design aligns with the "separating-all" configuration evaluated in §5.2.1.

Schema discovery improves data quality and query formulation by exposing user-facing schemas over unstructured property graphs. Various approaches have been proposed [14, 15, 44]. Lbath et al. [44] present a framework for inferring node/edge types, their hierarchies, and edge multiplicity constraints (e.g., many-to-one). GMMSchema [15] applies hierarchical clustering over labels and properties, recursively splitting label-based groups with a Gaussian Mixture Model to produce subtypes and a schema graph. Their goal is to expose the schema to users while TurboLynx keeps the schema internally to optimize performance. They are complementary to TurboLynx since they can be adopted in our *CGC*.

A substantial line of work [3, 12, 31, 47, 52] improves path-query performance by limiting the explosion of intermediate results, most notably through WCOJ and factorized query evaluation. WCOJ executes multiway joins in time proportional to the AGM bound, avoiding the large intermediates of binary joins, as demonstrated in EmptyHeaded [3] and in parallel settings [17]. Graphflow [47] introduces a cost model (intersection-cost), generates hybrid plans that mix binary joins with WCOJ-style intersections. More recently, Umbra's diamond hardened joins [12] split join processing into *Lookup* and *Expand* phases to pre-empt diamond-shaped blow-ups while preserving robust performance across relational and graph workloads. Complementarily, factorized representations compress repeated structure in intermediates and outputs. Graphflow's list-based processing [31] keeps factorized style adjacency lists, and F-IVM [52] maintains compressed views under updates.

7 CONCLUSION

In this paper, we presented TurboLynx, a novel graph database engine designed to enhance the performance of property graph data workloads. Our hypothesis was that state-of-the-art GDBMSs suffered from poor performance on complex queries, mainly due to the lack of proper handling of the schemaless nature in the PGM data. Addressing this challenge required us to rethink the architecture of the storage system, query processor, and query optimizer with schemaless as a primary design component. We introduced graphlet concepts to tame the high variability of schemas in PGM data and developed techniques for the query processor and optimizer to successfully mitigate the issues that followed. Our evaluation revealed that TurboLynx, integrating all our techniques, was highly effective and could deliver up to 183.9× improved performance.

ACKNOWLEDGMENTS

This work was partly supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2021-II210859, Development of a distributed graph DBMS for intelligent processing of big graphs, 90%) and supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2025-00517736, 10%).

REFERENCES

[1] 2025. https://github.com/duckdb/duckdb/pull/9751 Accessed: 2025-1-15.

[2] Lakshya A Agrawal, Nikunj Singhal, and Raghava Mutharaju. 2022. A SPARQL to cypher transpiler: Proposal and initial results. In *Proceedings of the 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD)*. 312–313.

[3] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[4] Renzo Angles. 2018. The property graph database model. In *AMW*.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 1421–1432.

[6] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. Pg-schema: Schemas for property graphs. In *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data (SIGMOD '23)*. 1–25.

[7] ArangoDB. 2024. https://arangodb.com/. Accessed: 2024-03-10.

[8] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[9] Nimo Beeren and George Fletcher. 2017. A Formal Design Framework for Practical Property Graph Schema Languages. In *Proceedings of the Conference on Extending Database Technology (EDBT)*. 478–484.

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 221–230.

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

[12] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust join processing with diamond hardened joins. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3215–3228.

[13] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Cidr*, Vol. 5. 225–237.

[14] Angela Bonifati, Stefania Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacôme Luton, and Thomas Pickles. 2022. DiscoPG: property graph schema discovery and exploration. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3654–3657.

[15] Angela Bonifati, Stefania Dumbrava, and Nicolas Mir. 2022. Hierarchical clustering for property graph schema discovery. In *EDBT 2022: 25th International Conference on Extending Database Technology*. OpenProceedings. org, 449–453.

[16] Yijian Cheng, Pengjie Ding, Tongtong Wang, Wei Lu, and Xiaoyong Du. 2019. Which category is better: benchmarking relational and graph database management systems. *Data Science and Engineering* 4, 4 (2019), 309–322.

[17] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 63–78.

[18] Marek Ciglan, Alex Averbuch, and Ladialav Hluchy. 2012. Benchmarking traversal operations over graph databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 186–189.

[19] DataStax. 2024. https://www.datastax.com/. Accessed: 2024-03-10.

[20] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248* (2019).

[21] DGraph. 2024. https://dgraph.io/. Accessed: 2024-03-10.

[22] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. 445–458.

[23] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 619–630.

[24] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[25] Leonidas Fegaras. 1998. A New Heuristic for Optimizing Large Queries. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA '98)*. Springer-Verlag, Berlin, Heidelberg, 726–735.

[26] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, Vol. 7. 25–35.

[27] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 1433–1445.

[28] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 599–613.

[29] SAP HANA Graph. 2024. https://help.sap.com/viewer/f381aa9c4b99457fb3c6b53a2fd29c02/latest/en-US. Accessed: 2024-07-11.

[30] José Rolando Guay Paz. 2018. Introduction to azure cosmos db. *Microsoft Azure Cosmos DB Revealed: A Multi-Model Database Designed for the Cloud* (2018), 1–23.

[31] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2491–2504.

[32] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. 2018. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 1789–1792.

[33] HGraphDB. 2024. https://github.com/rayokota/hgraphdb. Accessed: 2024-03-10.

[34] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting bugs in Cypher graph database engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. 163–174.

[35] Borislav Iordanov. 2010. Hypergraphdb: a generalized graph database. In *Web-Age Information Management: WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010 Revised Selected Papers 11*. Springer, 25–36.

[36] ISO/GQL. 2025. https://www.gqlstandards.org/. Accessed: 2025-03-22.

[37] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 745–761.

[38] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. {TEGRA}: Efficient {Ad-Hoc} analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 337–355.

[39] JanusGraph. 2024. https://janusgraph.org/. Accessed: 2024-03-10.

[40] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GRainDB: A Relational-core Graph-Relational DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*.

[41] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. 140–149.

[42] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

[43] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, et al. 2023. {GLogS}: Interactive graph pattern matching query at large scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 53–69.

[44] Hanâ Lbath, Angela Bonifati, and Russ Harmer. 2021. Schema inference for property graphs. In *EDBT 2021-24th International Conference on Extending Database Technology*. 499–504.

[45] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. 2530–2542.

[46] Memgraph. 2024. https://memgraph.com/. Accessed: 2024-07-12.

[47] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[48] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *International Semantic Web Conference*. Springer, 454–469.

[49] Neo4j. 2024. https://neo4j.com/. Accessed: 2024-03-10.

[50] Amazon Neptune. 2024. https://aws.amazon.com/neptune/. Accessed: 2024-03-10.

[51] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[52] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 365–380.

[53] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*. 1981–1984.

[54] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS. In *BICOD*. 10–19.

[55] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[56] Neo4j Pipelined Runtime. 2024. https://neo4j.com/docs/cypher-manual/current/planning-and-tuning/runtimes/concepts/#runtimes-pipelined-runtime. Accessed: 2024-03-10.

[57] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse-Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[58] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 505–516.

[59] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, et al. 2023. Bridging the Gap between Relational {OLTP} and Graph-based {OLAP}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 181–196.

[60] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 337–348.

[61] Oracle Spatial and Graph. 2024. https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html. Accessed: 2024-03-10.

[62] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1887–1901.

[63] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. 2014. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 815–826.

[64] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A Boncz. 2023. DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*.

[65] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting synergistic and retrofittable graph queries inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 345–359.

[66] TigerGraph. 2024. https://www.tigergraph.com/. Accessed: 2024-03-10.

[67] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, et al. 2021. {aDFS}: An Almost {Depth-First-Search} Distributed {Graph-Querying} System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 209–224.

[68] 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*. 1–6.

[69] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. 2024. {μSlope}: High Compression and Fast Search on {Semi-Structured} Logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 529–544.

[70] Xiating Xie, Xingda Wei, Rong Chen, and Haibo Chen. 2019. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 723–738.

[71] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 157–168.

[72] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. 2014. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment* 7, 7 (2014), 565–576.

[73] Zihao Zhao, Xiaodong Ge, Zhihong Shen, Chuan Hu, and Huajin Wang. 2023. S2CTrans: Building a bridge from SPARQL to Cypher. In *International Conference on Database and Expert Systems Applications*. Springer, 424–430.

[74] Zeying Zhu, Kan Wu, and Zaoxing Liu. 2023. Arya: arbitrary graph pattern mining with decomposition-based sampling. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1013–1030.