TurboLynx in Action: A Schemaless Graph Engine 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

Real-world property graphs are often *schemaless*: records sharing the same label may carry different subsets of properties, and new properties can emerge continuously as the data evolves. Such heterogeneity gives users a high degree of flexibility, but it also makes analytical queries difficult to optimize and execute efficiently.

This demonstration presents an interactive interface for exploring how TurboLynx processes such graphs through two representative query scenarios over a preloaded `DBpedia` instance. In the first scenario, users run a query that filters on a property present in only a subset of records and observe how early pruning reduces unnecessary scans, changes the physical plan, and lowers query latency. In the second scenario, users run a multi-hop query with aggregation and compare execution before and after TurboLynx's optimizations to observe how the system reduces plan-space blowup, shrinks large null-heavy intermediates, and improves end-to-end runtime. Together, the two scenarios let users observe how schema heterogeneity affects query optimization and execution, and how TurboLynx mitigates those effects.

PVLDB Reference Format:
Taesung Lee, Jaehyun Ha, Byungchul Tak, and Wook-Shin Han. TurboLynx in Action: A Schemaless Graph Engine for General-Purpose Analytics. PVLDB, 19(12): 4830 - 4833, 2026.
doi:10.14778/3827998.3828133

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

1 INTRODUCTION

Graph database management systems (GDBMSes) [5–7, 9] are widely used for highly connected data and relationship-centric queries. Among graph data models, the property graph model (PGM) [1] is popular because vertices and edges can carry arbitrary key-value attributes whose schemas need not be predefined. This gives users a high degree of flexibility, especially when the data model is not fully understood at the outset. However, that same flexibility makes analytical queries harder to optimize and execute efficiently.

*Corresponding authors

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828133

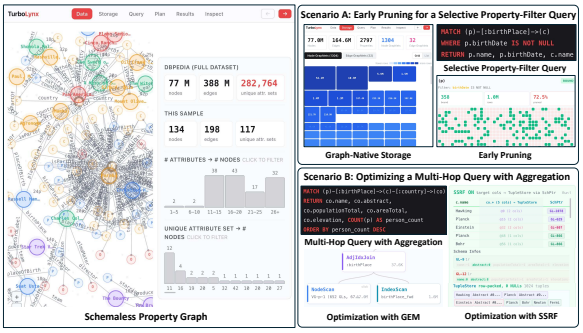


Figure 1: TurboLynx demo interface through two representative scenarios.

This demonstration presents TurboLynx [4], a graph analytics engine for general-purpose analytical processing on schemaless property graphs. Figure 1 provides an overview of the demo interface and its main interactive components. The demonstration is organized around two representative scenarios that highlight several challenges in schemaless graph processing. In Scenario A, users run a Cypher [3] query that filters on a property present in only a subset of records and observe how early pruning reduces unnecessary work, changes the physical plan, and lowers query latency. In Scenario B, users run a multi-hop query with aggregation and compare execution before and after TurboLynx's optimizations to see how the system controls optimization complexity, reduces intermediate overhead, and improves end-to-end runtime. Together, these scenarios show how schema heterogeneity affects query optimization and execution, and how TurboLynx mitigates those effects.

These scenarios are enabled by several components in TurboLynx. At the storage level, TurboLynx organizes heterogeneous records into columnar tables, which we call graphlets, that avoid both a single null-heavy wide table and excessive schema-specific fragmentation. A schema index supports early pruning for selective property predicates. During optimization, TurboLynx reduces complexity by grouping graphlet inputs before join enumeration. During execution, it uses compact shared-schema intermediates to reduce null overhead in join-heavy workloads. The rest of the paper briefly overviews these components and then shows how they appear in the interactive walkthrough.

2 SYSTEM ARCHITECTURE

TurboLynx consists of three main components: a *Graphlet Manager*, a *Graph Query Optimizer*, and a *Vectorized Graph Query Processor*

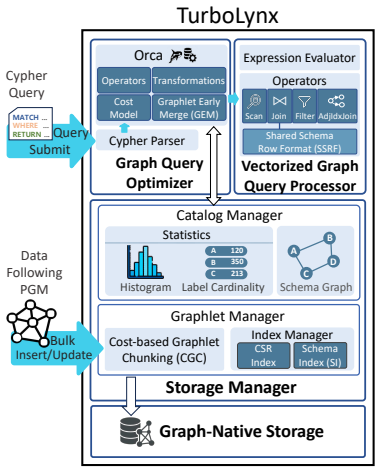


Figure 2: Architecture of TurboLynx.

(Figure 2). Together, these components support end-to-end analytical processing over schemaless property graphs. We briefly summarize these components below and refer the reader to [4] for full technical details.

Graphlet Manager. During bulk ingestion, the Graphlet Manager first groups vertices and edges by their label sets and then partitions each label-set group into *graphlets*, the basic storage units of TurboLynx. TurboLynx manages graphlets directly in its own graph-native storage layer, physically persisting each graphlet as a set of column files rather than relying on an external database system. The partitioning is driven by *Cost-based Graphlet Chunking* (*CGC*), which clusters records by attribute similarity and strikes a balance between two extremes: a separate table for every unique schema and a single wide schema. Its cost function jointly considers the number of graphlets, the number of null entries, and vectorization-related processing overhead. The resulting graphlets remain compact enough for efficient scans while avoiding the overheads of both extremes. As part of graphlet management, TurboLynx maintains two auxiliary index structures over the persisted graphlets: (i) a *compressed sparse row* (CSR) index for rapid neighborhood traversal, and (ii) a *schema index* (SI) for locating graphlets whose attribute sets match a query. The SI maps each attribute to the IDs of graphlets whose schemas contain that attribute, and multiple attribute predicates can be resolved by intersecting the corresponding graphlet-ID sets. As a result, TurboLynx can prune irrelevant graphlets before scanning while still supporting efficient graph traversal at query time.

Graph Query Optimizer. TurboLynx adapts the Orca optimizer [8] with graph-aware operators, transformation rules, and cost models. Its cost model refines scan costing at the graphlet level, using each graphlet's actual average tuple width rather than a single global row width. A key challenge is that each node label may correspond to multiple graphlets, and a pattern like `MATCH (a)-[]->(b)` compiles into a `UnionAll` over per-graphlet scans on each side. When `UnionAll` is pushed through joins, the optimizer must consider many logically equivalent alternatives, which can quickly enlarge the plan search space. *GEM* addresses this by grouping graphlets into coarse-granular virtual graphlets before join enumeration, so the optimizer sees fewer input tables without changing the physical data. This reduces the optimizer's search space while preserving opportunities for graphlet-specific optimization.

Vectorized Graph Query Processor. The query processor executes plans using specialized, vectorized operators over multi-schema inputs. A core challenge arises in binary joins, where intermediate schema combinations can grow quickly and introduce substantial null overhead. To address this issue, TurboLynx uses the *Shared Schema Row Format* (*SSRF*), a custom in-memory representation for heterogeneous join intermediates. *SSRF* separates schema metadata from tuple data and selectively combines columnar and row-based layouts. Sparse attributes are stored in row format, while base-column attributes remain in columnar format. *SSRF* stores each distinct intermediate schema once in a *SchemaInfo* table and stores sparse attribute values in a compact *TupleStore*, with per-tuple pointers via an *OffsetArr*. By storing schema definitions separately from the actual data, *SSRF* reduces schema management overhead, maintains low null-data handling costs, and avoids materializing an exhaustive set of schema combinations during join processing.

3 DEMONSTRATION

Our demonstration is organized around two representative scenarios over a preloaded DBpedia [2] instance. In the first scenario, attendees run a selective property-filter query and observe how early pruning changes the scanned graphlets, the resulting physical plan, and the end-to-end latency. In the second scenario, they run a multi-hop graph traversal query with aggregation and compare the execution times before and after TurboLynx's optimizations to assess how the system reduces optimizer and intermediate overhead. Together, these two workflows make the impact of schema heterogeneity on graph analytics directly visible.

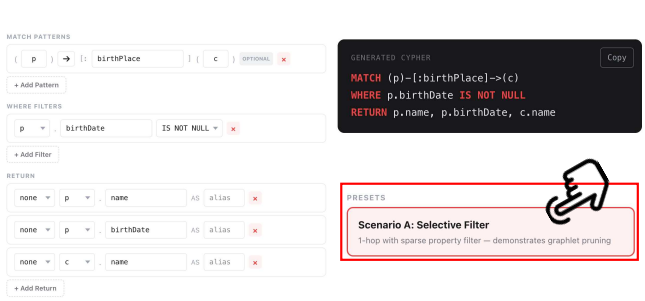
3.1 Scenario A: Early Pruning for a Selective Property-Filter Query

The first scenario centers on a query whose predicate refers to a property present in only a subset of records. It illustrates how graphlet-aware pruning reduces unnecessary scans. This scenario is organized into four steps.

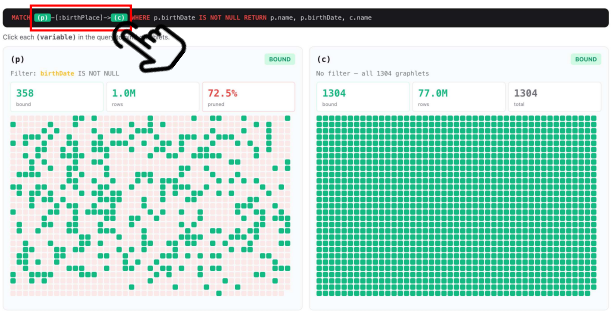
Step 1: Submit a selective property-filter query. Attendees begin with a representative query whose predicate targets a property present in only a subset of records, such as `WHERE p.birthDate IS NOT NULL`. The query view shows how this predicate maps to graphlet-level processing.

Step 2: Observe graphlet pruning. Once the predicate is applied, the interface highlights how the schema index resolves the requested property into graphlet-level retain/prune decisions: which graphlets contain the queried property and which are eliminated before scanning. The graphlet view updates immediately and exposes the reduced scan fanout. A metric panel summarizes the pruning effect: 359 of 1,304 graphlets remain, which means that 72% are pruned before scanning.

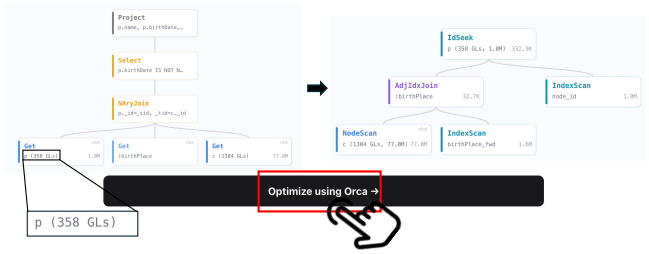
Step 3: Inspect the updated physical plan. The plan view then presents the optimized graphlet-aware plan after pruning, along with the remaining `UnionAll` inputs and the selected operators. A before/after comparison shows how early graphlet elimination



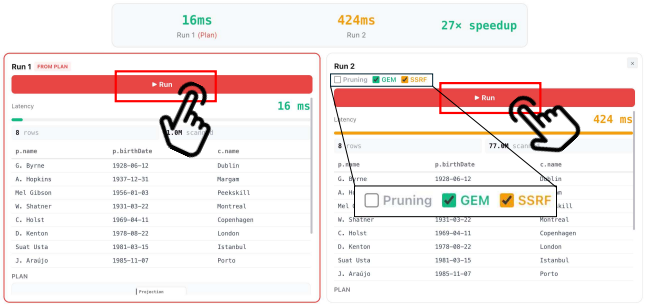
(a) Submit a selective property-filter query.



(b) Observe irrelevant graphlets being pruned.



(c) Inspect the updated physical plan.



(d) Run the query and observe the latency change.

Figure 3: Scenario A illustrates early graphlet pruning and plan-latency changes for a selective property-filter query.

changes the plan shape and reduces work. The panel also reports per-operator timing so that the reduction in scan fanout can be tied to the behavior of the final plan.

Step 4: Run the query and compare latency. The same query is then executed in the live query runner. The interface reports the returned tuples together with the end-to-end latency and highlights the effect of pruning at the workload level. For the preset query, the demo reports a latency reduction from 424 ms without pruning to 16 ms with pruning.

Taken together, these four steps show how a sparse property predicate prunes graphlets early, reshapes the physical plan, and lowers end-to-end runtime.

3.2 Scenario B: Optimizing a Multi-Hop Query with Aggregation

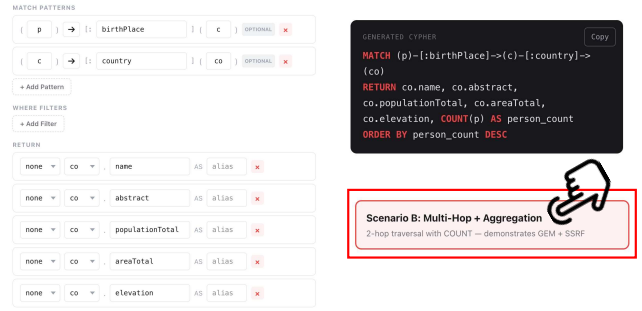
The second scenario focuses on a multi-hop query with aggregation. Here, schema heterogeneity affects both optimization and execution. Each node pattern may match multiple graphlets, and repeated joins may produce many heterogeneous intermediates. This scenario also proceeds in four steps.

Step 1: Submit a multi-hop query with aggregation. Attendees begin with a representative multi-hop query of the form `MATCH (a)-[]->(b)-[]->(c) ... RETURN ...`, where aggregation is expressed in the `RETURN` clause. This query serves as the basis for examining optimizer behavior, intermediate layout changes, and end-to-end execution.

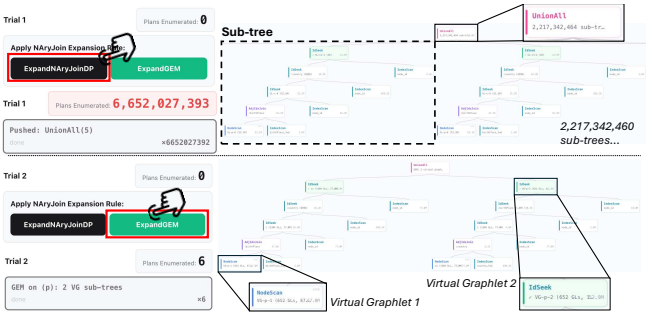
Step 2: Observe how *GEM* reduces optimizer complexity. The optimizer view explains why this type of query is difficult in schemaless graphs. Because each node pattern may correspond to multiple graphlets, the logical plan contains multiple `UnionAll` inputs. Once joins are pushed below these unions, the number of logically equivalent alternatives grows quickly. The interface reports this growth through an animated counter and a plan-space view. It then shows how *GEM* groups graphlets into virtual graphlets, so that the optimizer considers fewer graphlet-level inputs. The key summary number is the reduction in optimizer complexity: the number of candidate plans/input groups decreases from 6.6 billion to 6. This step demonstrates that TurboLynx does not merely choose a different final plan. It also reduces the search space required to find that plan.

Step 3: Run the query and compare latency. The same preset query is then executed end-to-end in the live query runner. The interface reports the result tuples, per-operator timings, and the final latency before and after TurboLynx's optimizations are enabled. For the preset query, the demo reports a latency reduction from 1,579 ms before optimization to 481 ms after optimization.

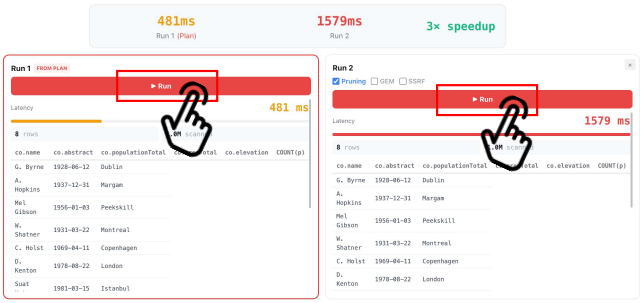
Step 4: Inspect intermediate layouts across joins. The walk-through then turns to the inspection of query execution. The inspection view presents the intermediate layouts produced by individual joins. For each join, the interface compares two representations for the resulting intermediate: a wide-table baseline and TurboLynx's shared-schema layout. This view shows how repeated joins increase both the number of schema combinations and the amount of null-heavy data under the wide-table representation. By contrast, the shared-schema layout keeps schema information separate and stores sparse values compactly. The comparison panel reports the core execution-side metrics for each join, including null reduction rates such as 10% for Join 1 and 57% for Join 2. These metrics highlight how much null-heavy data is removed by the shared-schema layout without requiring attendees to inspect the internal representation in detail.



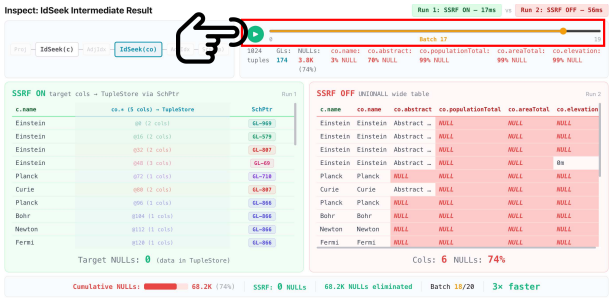
(a) Submit a multi-hop query with aggregation.



(b) Compare optimizer search space before and after *GEM*.



(c) Run the query and compare end-to-end latency.



(d) Inspect intermediate layouts across joins.

Figure 4: Scenario B illustrates optimizer and intermediate-overhead reduction for a multi-hop aggregation query.

These steps show how TurboLynx controls search-space blowup, reduces intermediate overhead, and improves end-to-end latency.

3.3 Live Operation Model and Audience Interaction

Full DBpedia is preloaded in the system, while a sampled subgraph is used only for load-time visualization. Attendees can run three vetted preset queries as well as a bounded custom mode, and can vary predicate and grouping choices to inspect how graphlet pruning and virtual-graphlet grouping change under different queries. All visual controls, including predicate changes, grouping changes, plan updates, and runtime counters, are updated in real time, allowing users to relate workload characteristics such as property sparsity and the number of graphlet-level inputs in multi-hop queries to optimizer behavior. If a custom query exceeds a predefined threshold, the interface falls back to precomputed traces to keep the demonstration responsive during live interaction. This operating model supports both guided walkthroughs of the two scenarios and self-directed exploration through selected controls.

4 CONCLUSION

We present TurboLynx, a graph analytics engine designed for property graphs with severe schema heterogeneity. Through an interactive web interface organized around two representative scenarios, attendees can see how selective property filters trigger early graphlet pruning and reshape the physical plan, and how multi-hop queries with aggregation benefit from reduced optimization overhead and more compact join intermediates. By linking these behaviors to visible changes in plan structure and end-to-end latency, the demonstration shows how graph-aware optimization and execution translate into practical performance gains.

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] Renzo Angles. 2018. The property graph database model. In *AMW*.
[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *International Semantic Web Conference*. Springer, 722–735.
[3] 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.
[4] Taesung Lee, Jaehyun Ha, Byungchul Tak, and Wook-Shin Han. 2026. TurboLynx: Schemaless graph engine strikes back for general-purpose analytics. *Proceedings of the VLDB Endowment* 19, 6 (2026), 1250–1263.
[5] Memgraph. 2024. https://memgraph.com/. Accessed: 2024-07-12.
[6] Neo4j. 2024. https://neo4j.com/. Accessed: 2024-03-10.
[7] Amazon Neptune. 2024. https://aws.amazon.com/neptune/. Accessed: 2024-03-10.
[8] 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.
[9] TigerGraph. 2024. https://www.tigergraph.com/. Accessed: 2024-03-10.