cuRPQ+: A System for Interactive Path-Aware Querying Beyond Plain CRPQs

Seohyeon Kim
KAIST
Daejeon, Republic of Korea
asdf0322@kaist.ac.kr

Sungwoo Park
KAIST
Daejeon, Republic of Korea
sungwoo.park@kaist.ac.kr

Min-Soo Kim*
KAIST
Daejeon, Republic of Korea
minsoo.k@kaist.ac.kr

ABSTRACT

Many graph analyses require reasoning about relationships between matched paths, but plain conjunctive regular path queries (CRPQs) cannot express such conditions directly. This demo presents *cuRPQ+*, a GPU-based engine and interactive system for extended CRPQs with three representative path constraints: word equality, prefix, and bounded length. Building on *cuRPQ*, *cuRPQ+* supports these constraints within a unified execution framework. The web interface lets users construct extended CRPQs, inspect sampled paths, compare plain and constrained results, and observe runtime and memory usage. Through these interactions, users can see how path constraints change query semantics and execution behavior. A demonstration video is available at https://youtu.be/V7Qtb9dsMTc.

PVLDB Reference Format:
Seohyeon Kim, Sungwoo Park, and Min-Soo Kim. cuRPQ+: A System for Interactive Path-Aware Querying
Beyond Plain CRPQs. PVLDB, 19(12): 4698 - 4701, 2026.
doi:10.14778/3827998.3828100

1 INTRODUCTION

A *Regular Path Query* (RPQ) is a core abstraction for querying edge-labeled graphs. It returns all distinct pairs of vertices (u, v) such that there exists a path from u to v whose edge-label sequence matches the given regular expression.

Conjunctive Regular Path Queries (CRPQs) extend RPQs by matching subgraph patterns in which multiple RPQ atoms must hold simultaneously. CRPQs are widely used in modern graph query languages for graph pattern matching.

However, CRPQs cannot directly express relationships among those paths [1]. Modern graph analyses often need to inspect and compare matched paths themselves, rather than only test whether a path satisfies a pattern [3–5]. For example, analysts may want to identify pairs of paths that share the same interaction sequence, examine whether a shorter trace is a proper prefix of a longer activity chain, or restrict analysis to short and interpretable paths. Such tasks require path-level constraints beyond plain CRPQs [1].

This need is not limited to social graphs. Path comparison also arises in semantic association analysis, approximate matching, and biological sequence analysis [1]. Similar needs appear in practice in fraud detection, abuse detection, and social-media or security-oriented graph analysis, where repeated or coordinated activity sequences can be informative [2, 6, 8, 9].

To support such queries, we develop *cuRPQ+*, a GPU-based engine for processing extended CRPQs. cuRPQ+ supports three useful path constraints: *word equality*, *prefix*, and *bounded length*. Efficient support is challenging because CRPQ evaluation is NP-complete in terms of combined computational complexity, and path constraints further require comparing or restricting matched paths during execution [6]. Building on our prior system, *cuRPQ*, cuRPQ+ extends high-performance RPQ/CRPQ processing to more expressive path queries [6].

Based on *cuRPQ+*, we present an interactive web interface for exploring ECRPQs. Users can construct constrained queries, inspect sampled matched paths, and compare result counts, runtime, and memory usage with their plain CRPQ counterparts. This allows users to observe both the semantic effect of path constraints and the execution cost of supporting them.

2 EXTENDED CRPQS (ECRPQS)

Let $G = (V, E, \lambda)$ be a directed edge-labeled graph, where $\lambda : E \to \Sigma$ assigns an edge label to each edge. For a path $P = e_1 \cdots e_k$, let $L(P) = \lambda(e_1) \cdots \lambda(e_k)$ denote its edge-label sequence and $|P| = k$ denote its hop length. An *RPQ* atom $x \xrightarrow{\rho} y$ returns endpoint pairs (u, v) connected by a path whose label sequence belongs to the language of the regular expression ρ. A *CRPQ* is a conjunction of RPQ atoms and returns tuples of data vertices assigned to the query variables. Following the ECRPQ framework of Barceló et al. [1], ECRPQs augment CRPQs with path variables and relations over path-label sequences. While RPQs and CRPQs primarily focus on vertices matched to query variables, ECRPQs can also refer to matched paths and constrain their label sequences or lengths. In this paper, we support a practical ECRPQ fragment with three representative constraints: word equality, prefix, and bounded length.

We use the path expression $(POSTS \mid LIKES) \rightarrow (HAS\_REPLY \mid HAS\_QUOTES)^*$ as a running example throughout this section. It matches paths that start with a *POSTS* or *LIKES* edge and then follow a sequence of *HAS_REPLY* or *HAS_QUOTES* edges. In Figure 1(a), the pink and blue paths correspond to the examples in Figure 1(b) and (c), respectively.

Word equality. Word equality requires two matched paths, P_1 and P_2, to have the same edge-label sequence, written as $L(P_1) = L(P_2)$.

In Figure 1(b), the selected paths share the same label sequence, $POSTS \rightarrow HAS\_REPLY \rightarrow HAS\_QUOTES$. Without word equality, a plain CRPQ only checks whether the regular pattern is satisfied between vertices, even if the matched paths have different label sequences, such as one beginning with *POSTS* and another with

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828100

LIKES. This makes word equality useful for identifying repeated interaction patterns.

Prefix. A prefix constraint requires the label sequence of one matched path to be a proper prefix of that of another. In particular, P_1 is a proper prefix of P_2 when $L(P_1) \prec L(P_2)$. In Figure 1(c), the shorter matched path forms a proper prefix of the longer matched path. This is useful for determining whether a shorter trace forms the beginning of a longer one.

Bounded length. A bounded length constraint restricts the hop length of a path, written as $\ell_{\min} \le |P| \le \ell_{\max}$. In Figure 1(c), we illustrate the special case of an upper-bound constraint, which filters out longer continuations and keeps only short local traces. For example, a longer candidate path such as $(u_2, p_2, c_4, c_5, c_6)$ is excluded because its length is 4. This is useful when users want concise and interpretable traces rather than arbitrarily deep cascades.

Together, these constraints allow users to reason about relationships among matched paths, forming the ECRPQ fragment supported by *cuRPQ+*.

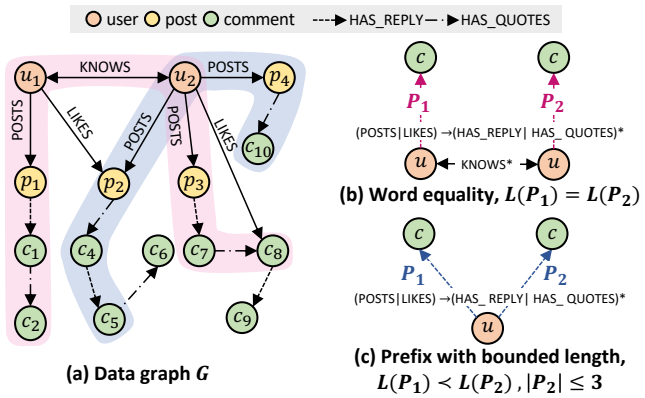


Figure 1: An example of ECRPQs.

3 THE CURPQ+ SYSTEM

This section summarizes the cuRPQ+ pipeline that powers the demonstration. At a high level, the engine extends the base cuRPQ workflow with lock-step traversal for word equality and prefix, together with bounded-length pruning, and returns both matches and sampled paths to the interface.

cuRPQ+ is not merely a front-end for cuRPQ. The original cuRPQ system supports high-performance evaluation of RPQs and plain CRPQs, where each RPQ atom is evaluated and the resulting vertex pairs are joined. cuRPQ+ extends this pipeline in three ways. First, it supports a practical fragment of ECRPQs that explicitly constrains matched paths using word equality, prefix, and bounded length. Second, it adds constraint-aware execution operators, including lock-step traversal for word equality and prefix constraints and bounded-depth pruning for length constraints. Third, it exposes the semantic effect of these constraints through an interactive interface that compares the unconstrained CRPQ with the corresponding constrained query and shows representative matched paths.

3.1 cuRPQ+ Architecture

Figure 2 shows the overall cuRPQ+ architecture. The web interface submits a JSON query to the engine, which compiles the query and executes GPU traversal over partitioned graph data. Standard RPQ atoms use the RPQ traversal pipeline, word equality and prefix use lock-step traversal, and bounded length is enforced during planning. The engine then joins partial matches and returns sampled paths and execution results to the result viewer.

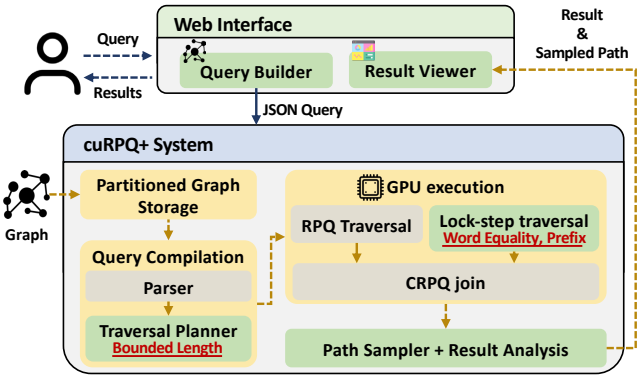


Figure 2: The architecture of the cuRPQ+ system. Components shown in green indicate parts that are extended or newly introduced in cuRPQ+.

3.2 Base RPQ/CRPQ Processing

Partitioned Graph Storage. Since real-world graphs often exceed GPU memory, cuRPQ stores the graph in Labeled Grid Format (LGF) [6, 7], a partitioned layout that loads only the graph partitions required by the current traversal. This allows cuRPQ+ to process graphs that do not fit entirely in GPU memory.

Query compilation. Each RPQ atom is compiled into a finite automaton and translated into an automaton-guided traversal plan. The plan restricts traversal to the graph partitions and edge labels relevant to the query.

RPQ traversal. At runtime, cuRPQ explores the traversal plan on the GPU in parallel from many source vertices. At each step, it follows only edges valid for the current automaton state and activates only the graph partitions selected by the current plan node; whenever a final automaton state is reached, the corresponding result is recorded.

CRPQ join. For a CRPQ, cuRPQ first evaluates each RPQ atom independently and then joins their results to produce full CRPQ results. Throughout this process, GPU memory remains bounded by using fixed-size buffers and overlapping CPU-side materialization with continued GPU exploration.

3.3 Path Constraint Extensions

Figure 2 highlights in green the components extended in cuRPQ+ for path constraints. In particular, the traversal planner is extended to support bounded-length constraints, and GPU execution is augmented with lock-step traversal for word equality and prefix.

Traversal Planner. A bounded-length constraint restricts the range of valid hop counts. To support this constraint, cuRPQ+ extends the traversal planner so that paths exceeding the allowed depth are pruned, and only permitted depths are marked as terminal. In Figure 3(b), we illustrate the special case of an upper-bound

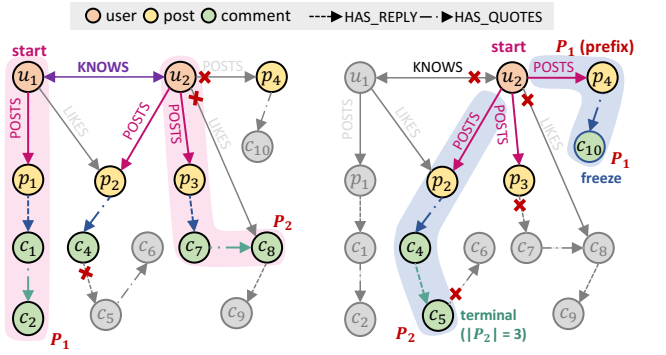
constraint. The traversal for P_2 continues only up to the bounded-length limit and stops once the path reaches the maximum allowed length.

Lock-step Traversal. Figure 3 conceptually illustrates the lock-step traversal used for word equality and prefix. In the actual system, this behavior is realized through automaton-guided traversal plans rather than explicit query-graph traversal. The two paths are evaluated jointly while consuming the same labels, so mismatching candidates are pruned immediately during traversal.

For *word equality*, cuRPQ+ requires two matched paths to have exactly the same edge-label sequence. Enumerating candidate paths independently and comparing them afterward is prohibitively expensive when many paths exist. Instead, cuRPQ+ evaluates the two paths jointly via lock-step traversal. At each hop, both traversals advance only on edges with the same label, ensuring that every completed joint traversal already satisfies the constraint. For example, in Figure 3(a), u_1 and u_2 are the source vertices of the compared paths, and the pair (u_1, u_2) satisfies the *KNOWS* atom in the motivating query. The two traversals then proceed hop by hop only when they consume the same label.

For *prefix*, cuRPQ+ requires the label sequence of one path to be a proper prefix of another. It uses the same joint traversal while both paths continue to match the same labels. Once the shorter path P_1 reaches a terminal state, cuRPQ+ freezes P_1 and continues traversal only for P_2. A result is produced only if P_2 extends beyond that point and reaches a valid final state. Figure 3(b) illustrates this behavior. As in word equality, the two traversals first advance in lock step while consuming the same labels. Once P_1 satisfies its path condition, it is frozen, while traversal continues only for P_2.

Path Sampler and Result Analysis. After query execution, cuRPQ+ extracts sampled result paths and summary execution statistics for presentation in the interface.



(a) Word equality ($L(P_1) = L(P_2)$)

(b) Prefix constraint + length constraint ($L(P_1) \prec L(P_2)$, $|P_2| \leq 3$)

Figure 3: Lock-step traversal for word equality and prefix.

4 DEMONSTRATION SCENARIOS

Path-constrained queries help analyze interaction traces, such as repeated activity sequences or short local cascades. The demonstration lets users add word-equality, prefix, or bounded-length constraints to plain RPQs/CRPQs and observe their effects on matched paths, result counts, runtime, and memory usage.

4.1 End-to-End Experience

The demonstration provides a web-based workflow for exploring, executing, and inspecting extended CRPQs over graph datasets.

Step 1. Dataset selection. Users first select a dataset and scale factor. The demonstration supports multiple datasets, including LDBC SNB and StackOverflow, each at several scales.

Step 2. Scenario selection. From the home screen, users may either run a guided scenario or build a query from scratch. Users may choose either a plain RPQ/CRPQ or a constraint-extended query, with guided scenarios provided for representative cases.

Step 3. Query construction and editing. Users construct queries through an interactive query-graph builder. They may create a plain RPQ/CRPQ, extend it with path constraints, or modify an existing template. For advanced use, queries can also be edited in JSON form.

Step 4. Sampled path inspection. After execution, the default *Graph* view shows sampled result paths with a path list. It also displays the total match count and GPU execution time at the bottom. This view helps users inspect not only whether matches exist, but also how the returned paths satisfy the intended path relationship.

Step 5. Result views. The interface provides multiple result views for inspection and comparison. The comparison-oriented views highlighted in Figure 4 are described in detail in Scenarios 1 and 2.

4.2 Scenario 1 – Semantic Differences Beyond Plain CRPQs

The first scenario uses the Compare view to explain the semantic effect of adding a path constraint. For a selected ECRPQ, cuRPQ+ also runs the corresponding plain CRPQ obtained by removing the path constraint while keeping the same RPQ atoms. The two results are shown side by side, and sampled paths filtered by the constraint explain the difference. For word equality, for example, two paths may both match the same regular expression but have different label sequences; the ECRPQ rejects such pairs because $L(P_1) \neq L(P_2)$. Prefix and bounded-length scenarios similarly show paths filtered because the shorter path is not a prefix or because the path exceeds the length bound.

4.3 Scenario 2 – System Efficiency Comparison

The second scenario shows the execution cost of evaluating the same path-constrained query under different backends. For a fixed query and dataset configuration, the Benchmark view compares the runtime and peak memory usage of cuRPQ+ with those of CPU-based systems such as Umbra and DuckDB. It also reports execution status, including timeout and out-of-memory outcomes, for the same query. In the live demo, baseline systems are executed on demand under fixed timeout and memory budgets.

Settings and queries. We evaluate cuRPQ+ on LDBC SNB (SF=1 and 10), comparing against DuckDB and Umbra under the same SQL-encoded query semantics on a server with two AMD EPYC 7302 CPUs, 1 TB RAM, and an NVIDIA A100 GPU. The workload consists of five representative ECRPQs, each combining CRPQ structural joins with a path-relational constraint:

- **Q1** (word equality): $L(P_1)$=$L(P_2)$, $P_1 \in a^*$, $P_2 \in a \cdot a$.
- **Q2** (word equality): $L(P_1)$=$L(P_2)$, $P_1, P_2 \in a^* \cdot b \cdot c$.

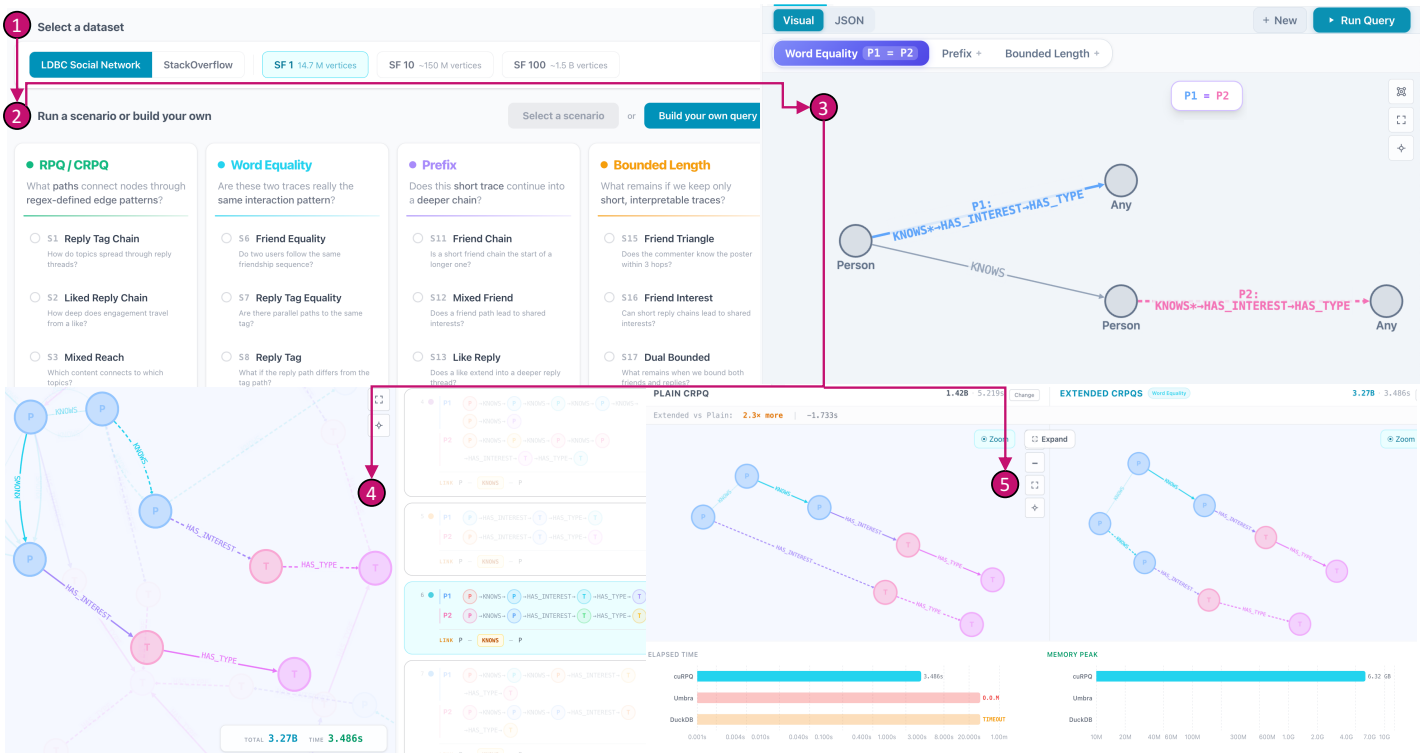


Figure 4: End-to-end workflow of the cuRPQ+ demonstration.

- **Q3** (prefix): $L(P_1) \prec L(P_2)$, $P_1 \in a \cdot a$, $P_2 \in a \cdot a \cdot a$.
- **Q4** (prefix): $L(P_1) \prec L(P_2)$, $P_1 \in a^* \mid b$, $P_2 \in a \cdot a$.
- **Q5** (bounded length): $1 \leq |P| \leq 3$ on a.

Native graph systems are excluded because word equality and prefix constraints are not expressible in their query languages.

Performance comparison. Table 1 reports end-to-end runtimes. On SF=1, only cuRPQ+ completes all five queries, achieving up to 131× speedup on Q5. On SF=10, both baselines fail on every query while cuRPQ+ completes all within 750 s. On Q5 (SF=1), cuRPQ+ also uses only 6.7 GB of memory versus 169–171 GB for the baselines.

Table 1: Comparison of execution times (in sec). (T.O.: elapsed time exceeds 1 hour, O.O.M.: out of memory).

SF	System	Q1	Q2	Q3	Q4	Q5
1	DuckDB	T.O.	T.O.	T.O.	T.O.	157.2
	Umbra	T.O.	O.O.M.	O.O.M.	O.O.M.	13.0
	cuRPQ+	**1.7**	**5.2**	**3.1**	**2.0**	**1.2**
10	DuckDB	T.O.	T.O.	T.O.	T.O.	O.O.M.
	Umbra	O.O.M.	O.O.M.	O.O.M.	O.O.M.	O.O.M.
	cuRPQ+	**48.2**	**746.4**	**74.1**	**44.3**	**9.3**

5 CONCLUSION

In this demonstration, we presented cuRPQ+, a GPU-based engine and interactive system for querying beyond plain CRPQs. We highlighted that plain CRPQs cannot directly express path relations such as word equality, prefix, and bounded length, while cuRPQ+ supports them within a unified execution framework for extended CRPQs. Through interactive scenarios, users can compare constrained queries with their plain counterparts, inspect sampled result paths, and observe cross-system performance differences. Overall, this demonstration highlights both the expressive benefits of extended CRPQs and the efficient processing of such queries in cuRPQ+.

REFERENCES

[1] Pablo Barceló, Leonid Libkin, Anthony W Lin, and Peter T Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)* 37, 4 (2012), 1–46.

[2] Federico Cinus, Marco Minici, Luca Luceri, and Emilio Ferrara. 2025. Exposing cross-platform coordinated inauthentic activity in the run-up to the 2024 us election. In *Proceedings of the ACM on Web Conference 2025*. 541–559.

[3] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proc. ACM SIGMOD International Conference on Management of Data*. 2246–2258.

[4] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1798–1810.

[5] Leonid Libkin, Wim Martens, Filip Murlak, Liat Peterfreund, and Domagoj Vrgoč. 2025. Querying graph data: Where we are and where to go. In *Companion of the 44th Symposium on Principles of Database Systems*. 9–26.

[6] Sungwoo Park, Seohyeon Kim, and Min-Soo Kim. 2026. cuRPQ: A High-Performance GPU-Based Framework for Processing Regular and Conjunctive Regular Path Queries. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD (2026), 1–28.

[7] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2025. cuMatch: A GPU-based Memory-Efficient Worst-case Optimal Join Processing Method for Subgraph Queries with Complex Patterns. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[8] Hadar Rotschield and Liat Peterfreund. 2025. On the Expressiveness of Languages for Querying Property Graphs in Relational Databases. *Proceedings of the ACM on Management of Data* 3, 5 (2025), 1–18.

[9] Karishma Sharma, Yizhou Zhang, Emilio Ferrara, and Yan Liu. 2021. Identifying coordinated accounts on social media through hidden influence and group behaviours. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 1441–1451.