cuRPQ: A High-Performance GPU-Based Framework for Processing Regular and Conjunctive Regular Path Queries

SUNGWOO PARK, Korea Advanced Institute of Science and Technology, Republic of Korea
SEOHYEON KIM, Korea Advanced Institute of Science and Technology, Republic of Korea
MIN-SOO KIM*, Korea Advanced Institute of Science and Technology, Republic of Korea

Regular path queries (RPQs) are fundamental for path-constrained reachability analysis, and more complex variants such as conjunctive regular path queries (CRPQs) are increasingly used in graph analytics. Evaluating these queries is computationally expensive, but to the best of our knowledge, no prior work has explored GPU acceleration. In this paper, we propose cuRPQ, a high-performance GPU-optimized framework for processing RPQs and CRPQs. cuRPQ addresses the key GPU challenges through a novel traversal algorithm, an efficient visited-set management scheme, and a concurrent exploration-materialization strategy. Extensive experiments show that cuRPQ outperforms state-of-the-art methods by orders of magnitude, without out-of-memory errors.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Regular path query, Conjunctive regular path query, GPU computing

ACM Reference Format:
Sungwoo Park, Seohyeon Kim, and Min-Soo Kim. 2026. cuRPQ: A High-Performance GPU-Based Framework for Processing Regular and Conjunctive Regular Path Queries. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 156 (June 2026), 28 pages. https://doi.org/10.1145/3802033

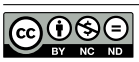
1 Introduction

A *Regular Path Query* (RPQ) is one of the most expressive and powerful graph queries [28, 79], which finds all pairs of vertices connected by a path whose sequence of edge labels matches a given regular expression. RPQs have two key characteristics: they allow paths of **arbitrary length** and return **distinct vertex pairs** [6, 52]. For example, given a data graph G with vertex and edge labels as in Figure 1, an RPQ Q_1 is the task of finding vertex pairs that satisfy $u_0 \xrightarrow{abc^*} u_1$. It returns a total of 13 results[1], among which $\{(v_2, v_2), (v_0, v_7), (v_0, v_{11})\}$ correspond to the paths *ab*, *abc*, and *abccc*, respectively. Q_1 represents an *all-pairs RPQ*, where both endpoints are variables, whereas the RPQ can also be expressed as a *single-source RPQ* when one of the endpoints is fixed to a constant vertex [12, 21]. RPQs are applied in various domains that require reachability analysis with path constraints, such as social network analysis [44], fraud detection [86], network security [68], and bioinformatics [47, 76]. Given their widespread applications, many graph query languages — such as openCypher [36], SPARQL [78], and ISO/GQL [23] — support RPQs [33, 64] as a fundamental operation.

*Corresponding author.

[1] $\{(v_0, v_1), (v_0, v_4), (v_0, v_7), (v_0, v_8), (v_0, v_9), (v_0, v_{10}), (v_0, v_{11}), (v_0, v_{12}), (v_0, v_{13}), (v_2, v_2), (v_2, v_3), (v_7, v_2), (v_7, v_3)\}$

Authors' Contact Information: Sungwoo Park, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, sungwoo.park@kaist.ac.kr; Seohyeon Kim, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, asdf0322@kaist.ac.kr; Min-Soo Kim, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, minsoo.k@kaist.ac.kr.



ACM 2836-6573/2026/6-ART156
https://doi.org/10.1145/3802033

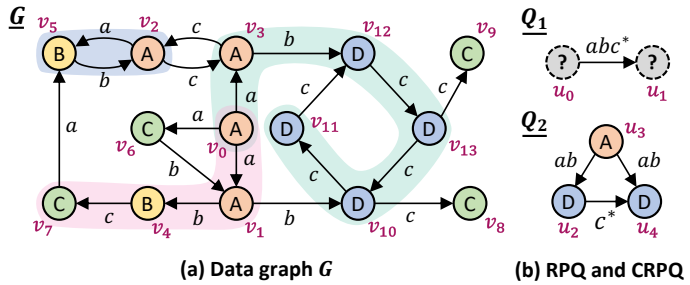


Fig. 1. Example of RPQ and CRPQ on a data graph.

Recently, as analyses required in various applications have become more complex [25, 26, 69, 73, 86], the demand for *Conjunctive Regular Path Queries* (CRPQ) [19] has increased. A CRPQ finds subgraph patterns that satisfy the regular expression constraints of multiple RPQ atoms simultaneously. In Figure 1, Q_2 on (u_2, u_3, u_4) maps to $\{(v_{10}, v_0, v_{10}), (v_{10}, v_0, v_{12}), (v_{12}, v_0, v_{10}), (v_{12}, v_0, v_{12})\}$. Representatively, CRPQs are applied in information propagation analysis to trace the creator User (u) and the related Post (p) of a Message (m) with a specific Tag (t:*Sports*) [73]. This can be expressed as a CRPQ that simultaneously satisfies three RPQ atoms: $m \xrightarrow{hasTag} t{:}Sports$, $m \xrightarrow{hasCreator} u$, and $m \xrightarrow{replyOf^*} p$. The second and third atoms correspond to all-pairs RPQs, since both endpoints are variables, and such all-pairs RPQs are frequently evaluated in CRPQs [73]. In general, a CRPQ is processed by evaluating each RPQ atom and then combining the results using joins or a *worst-case optimal multi-way join* (WCOJ) [20, 30, 41, 59, 77]. However, when evaluating all-pairs RPQs with densely connected edge labels, such as the *replyOf* relation among messages [75], the computational cost can become prohibitively high, since paths between almost all vertex pairs must be considered.

Most existing studies have addressed RPQs using two main approaches: (1) the algebra-based approach and (2) the automata-based approach [7, 8, 14, 35, 55, 64, 87]. The algebra-based approach [58, 70] computes the transitive closure (e.g., c^*) by repeatedly applying the α-operator [4], which iteratively performs join operations until a fixed point is reached. Other regular expression operators, such as concatenation (e.g., ab) and alternation (e.g., $a + b$), are handled through relational algebra operations such as join, union, and distinct. In contrast, the automata-based approach [7, 35, 45, 60, 62] simultaneously performs data-graph traversal and automaton transitions corresponding to the given regular expression. For each starting vertex, a *visited set* is maintained, storing pairs of data graph vertices and automaton states to ensure that only non-duplicate paths are explored.

In general, the computational complexity of all-pairs RPQ evaluation is *cubic* in the number of vertices for algebra-based methods, *quadratic* for automata-based methods, and *NP-complete* for CRPQs [1, 5, 9–11, 17]. These complexities represent a major bottleneck, motivating the use of GPUs to exploit their massive parallelism. Although GPU-based acceleration has been extensively studied for graph query processing [38, 46, 66, 72, 82, 84, 88, 89], to the best of our knowledge, no prior work has proposed a GPU-based method for RPQ processing. Designing such a GPU-based method for RPQ and CRPQ evaluation introduces **three fundamental challenges**.

First, naive GPU-based methods for arbitrary-length RPQs can suffer from severe *performance degradation* and may produce *incorrect answers* due to *traversal path length limitations*. Because of limited GPU memory, most GPU-based graph processing methods adopt DFS-based or DFS-like traversal strategies [13, 46, 49, 61, 66, 72, 82, 84, 88]. However, DFS inherently explores long paths first, often checking reachability inefficiently by traversing unnecessarily deep paths. On GPUs,

where only a portion of the graph can be loaded at a time, traversal depth is strictly limited, and DFS may easily exceed this limit, failing to discover valid paths.

Second, naive GPU-based methods significantly *reduce parallelism as the visited set quickly exceeds* GPU memory capacity. Achieving high performance on GPUs requires simultaneous traversal from thousands of starting vertices, but the size of the visited set grows proportionally with the number of starting vertices and can easily surpass available GPU memory [7]. This constraint prevents large-scale parallel traversal, reducing GPU utilization and thereby degrading overall performance.

Third, naive GPU-based methods *cannot evaluate CRPQs, as the materialization of RPQ results often exceeds* GPU memory capacity, and managing result explosion is especially challenging for all-pairs RPQs. In addition, the execution time of an all-pairs RPQ can vary greatly depending on optimization strategies (e.g., join ordering and materialized views in algebra-based approaches). To enable such optimizations in automata-based approaches, it is necessary to employ state-of-the-art plan representations such as WavePlan [87], which extend automaton-based plans, but still require the materialization of sub-RPQs, whose results can grow explosively. This problem also occurs in GPU-based DBMSs that support materialization [66]; for instance, the state-of-the-art GPU-based DBMS HeavyDB [40] and GPU library RAPIDS [18] are only feasible when the materialized RPQ results fit entirely within GPU memory.

To address the above challenges, we propose **cuRPQ, a high-performance GPU-based framework** for processing RPQs and CRPQs. It efficiently handles computationally expensive queries, such as all-pairs RPQs through a series of GPU-oriented optimizations, and can be seamlessly extended to support CRPQs. For the first challenge, we propose a **hop-limited level-wise DFS** method that performs DFS within fixed-hop ranges and advances the search in a level-wise manner. This design preserves the low memory footprint of DFS while achieving BFS-like progression across levels. For the second challenge, we propose an **on-demand segment pooling** technique that manages the visited set using fine-grained segments. This enables dynamic memory allocation and release, sustaining large-scale parallel exploration without exhausting GPU memory capacity. For the third challenge, we propose a **concurrent exploration-materialization** strategy for heterogeneous GPU-CPU environments, where GPUs focus on path exploration while CPUs concurrently perform GPU-aware graph materialization of large RPQ results. Through these techniques, cuRPQ effectively addresses the key challenges of GPU-based RPQ and CRPQ processing, fully exploiting GPU parallelism to achieve high performance and scalability.

The main contributions of this paper are as follows:

- We propose cuRPQ, a GPU-optimized framework for processing RPQ/CRPQ queries.
- We propose the hop-limited level-wise DFS, a novel traversal method for efficiently exploring arbitrary-length paths while maintaining low memory overhead.
- We propose the on-demand segment pooling technique, a fine-grained memory management technique that alleviates GPU memory bottlenecks in managing the visited set.
- We propose the concurrent exploration-materialization strategy that fully exploits GPU parallelism beyond GPU memory limits.
- Extensive experiments show that cuRPQ outperforms state-of-the-art CPU-based algebra and automata methods by up to 4,945X and 269X, respectively, even for queries producing trillions of results.

2 Preliminaries

2.1 Problem Definition

We consider a directed graph $G = (V, E, L)$ in which both vertices and edges are labeled. Here, V denotes the set of vertices, E the set of edges, and L the labeling function that assigns labels to

vertices and edges. We denote the label of a vertex v_i by $L(v_i)$ and the label of an edge (v_i, v_j) by $L(v_i, v_j)$. We define RPQ in Definition 2.1 [7, 8, 35, 62].

Definition 2.1. **Regular Path Query (RPQ)**: Given a data graph $G = (V, E, L)$, an RPQ is denoted as $x \xrightarrow{\rho} y$, where x and y are query variables that can take values from V, and ρ is a regular expression over the set of edge labels of G. The result of the RPQ is the set of all pairs $(x, y) \in V \times V$ such that there exists a path $p = (v_i \rightarrow v_{i+1} \rightarrow \cdots \rightarrow v_j)$ in G whose edge-label sequence $L(v_i, v_{i+1})\, L(v_{i+1}, v_{i+2}) \cdots L(v_{j-1}, v_j)$ matches the regular expression ρ, where $x = v_i$ and $y = v_j$.

By definition, an RPQ returns distinct (start, end) vertex pairs connected by at least one path matching a given regular expression. This pair-based semantics is standard in the RPQ literature and sufficient for many practical use cases that primarily focus on reachability [53]. A CRPQ generalizes RPQs by allowing multiple RPQ atoms to be combined in a conjunction. We define CRPQ in Definition 2.2 [30–32].

Definition 2.2. **Conjunctive Regular Path Query (CRPQ)**: Given a data graph $G = (V, E, L)$ and $Q = (V_q, E_q, L_q)$, where each edge in E_q is of the form $x \xrightarrow{\rho} y$ with $x, y \in V_q$ and a regular expression ρ. The result of the CRPQ is the set of all possible homomorphisms $f : V_q \rightarrow V$ such that (1) $\forall u \in V_q, L_q(u) = L(f(u))$; and (2) $\forall (x \xrightarrow{\rho} y) \in E_q$, there exists a path from $f(x)$ to $f(y)$ in G whose edge-label sequence matches ρ.

2.2 Processing RPQs

Graph pattern matching, also known as a *Conjunctive Query* (CQ), focuses on structural correspondences between a given pattern and a subgraph of a data graph [6, 14]. In contrast, an RPQ is a path query that supports transitive closure operators such as the Kleene star ($*$), enabling the exploration of arbitrary-length paths, and it returns only distinct vertex pairs even when multiple paths exist between them.

Algebra-based approach [58, 70]: This approach employs the α-operator [4], commonly expressed through the `WITH RECURSIVE` clause. In RPQs, this operator is used to compute transitive closure (e.g., c^*). The computation starts from the relation consisting of edges with the given label (e.g., c in c^*) and repeatedly joins it to expand reachable vertex pairs one hop at a time. The newly discovered pairs are merged with the accumulated set of reachable pairs through a union operation to preserve distinct results, and the process continues until no further pairs are discovered. Non-transitive edge patterns are processed by standard join operations. When several relations correspond to an edge label, they are first unified before processing. For example, in Q_1 of Figure 1, suppose there exist relations corresponding to edge labels a, b, and c. The relation for edge label a consists of tuples $\{(v_0, v_1), (v_0, v_3), (v_0, v_6), (v_2, v_5), (v_7, v_5)\}$. Here, c^* is computed as intermediate data through the α-operator, and the final results can then be derived by applying join and distinct operations with the relations corresponding to a and b.

Automata-based approach [7, 35, 45, 60, 62]: This approach exploits the automaton corresponding to the regular expression of the given RPQ. Traversal starts from the possible starting vertices in the data graph (e.g., all vertices in an all-pairs RPQ) and the initial state of the RPQ automaton, simultaneously performing automaton transitions and edge exploration. Whenever the automaton reaches a final state, the corresponding start-end vertex pair is added to the RPQ result set. Conceptually, this process can be viewed as operating over the *product graph* [7, 35, 62], in which each vertex corresponds to a pair consisting of a data-graph vertex and an automaton state. For each starting vertex, a *visited set* records explored product graph vertices to avoid redundancy and ensure distinct results. For example, Figure 2 presents the automata-based evaluation of Q_1 on

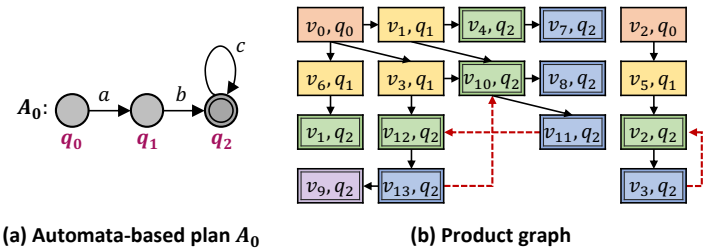


Fig. 2. Example of automata-based plan and product graph.

the data graph G shown in Figure 1. Figure 2(a) shows the automata plan corresponding to abc^*, while Figure 2(b) depicts a portion of the product graph derived from starting at v_0 and v_2. The red dashed lines indicate edges where traversal terminates because the corresponding product graph vertex is already in the visited set. In the traversal process, the vertices marked with double squares corresponding to 2-hop (green), 3-hop (blue), and 4-hop (purple) are included in the RPQ result set.

2.3 WavePlan

WavePlan [87] is a plan representation that extends the automata-based plan while incorporating the advantages of the algebra-based approach. The algebra-based approach materializes the results of transitive closure (e.g., c^*), which enables the reuse of partial paths and allows for further optimizations such as join ordering [87]. In contrast, the automata-based approach follows automaton state transitions and directly passes results to subsequent operations, enabling query processing in a *pipelined fashion* [87]. WavePlan primarily represents queries in an automata-based plan, while also allowing sub-RPQ results to be materialized for reuse and supporting diverse exploration orderings as needed. Figure 3 shows several WavePlans for abc^*. Beyond the A_0 plan from Figure 2, WavePlan supports the *reverse* plan A_1, which uses not only out-edges (e.g., $a\cdot$) but also in-edges (e.g., $\cdot a$); the *loop-cache* plan A_2, which reuses results from other automata; and the *start-in-the-middle* plans A_3 and A_4, which begin traversal from the middle of the path expression.

Finally, WavePlan is a plan representation that integrates the strengths of the automata-based and algebra-based approaches through an automata-based execution plan. By adopting an automata-based design, cuRPQ naturally integrates with WavePlan and can leverage execution-plan-level optimizations, even in GPU environments with limited memory capacity.

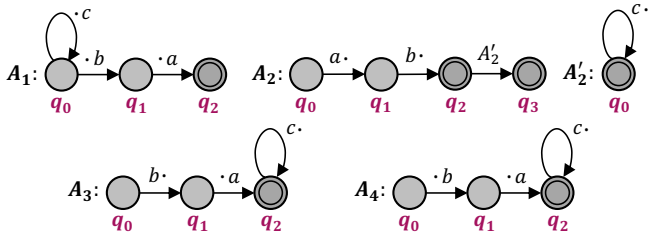


Fig. 3. Example of WavePlans.

2.4 Labeled Grid Format

The *Labeled Grid Format* (LGF) [66] extends the conventional 2D grid format [43, 50, 51, 61, 80, 90] — which partitions source and destination vertex ranges into fixed widths and heights — from simple graphs to labeled graphs. In LGF, a separate grid is maintained for each edge label, and the source and destination vertex ranges of each grid are logically partitioned based on vertex labels. Each partition, called a *block*, contains edges sharing the same source label, destination label, and edge label. LGF maintains block-level metadata called the *GridMap*. GridMap follows a 3D array

structure, each element in GridMap has coordinates (x, y, z), where x is the *block row ID* for source vertex, y is the *block column ID* for destination vertex, and z is the *grid ID* for edge label. LGF also maintains a *VertexLabel* table that stores vertex label names and their block-level ranges, and an *EdgeLabel* table that stores edge label names. Each block can be recursively partitioned into four quadrants along the source and destination dimensions, producing smaller *slices* until their sizes fall below a specific threshold [61, 66]. This guarantees that all partitions (i.e., slices) remain below the threshold, preventing any single slice from growing too large to fit in GPU memory. Each slice is stored in an adjacency-list representation (e.g., Compressed Sparse Row [34], trie-based index), and distinct slices are maintained for out-edges and in-edges to support bidirectional traversal. Finally, LGF can operate efficiently in both in-memory and disk-based environments [61, 66].

Figure 4(a) shows an example of LGF for the data graph G in Figure 1, which consists of four vertex labels $\{A, B, C, D\}$ and three edge labels $\{a, b, c\}$. For simplicity, all vertices with the same vertex label are assumed to fit within a single block (P_i) (i.e., both block width and height are 4), and each block is assumed to consist of a single slice (S_i). Thus, all ranges in the VertexLabel table have a length one (e.g., [0, 1)). Including in-edge slices, there are 24 blocks and slices in total, but due to space constraints only $GridMap_{out}$ for out-edge slices is shown. The edges stored in each slice are listed in Table 1. P_5, pointed to by $(0, 3, 1)$ in $GridMap$, corresponds to a single slice S_5 and contains two edges $\{(v_1, v_{10}), (v_3, v_{12})\}$ whose source vertex label is A, destination vertex label is D, and edge label is b.

In this paper, we adopt the LGF format for processing RPQs and CRPQs. In the automata-based approach, efficient traversal by edge label is required. LGF supports this by maintaining a separate grid for each label, enabling direct access to the edge set corresponding to a given edge label (i.e., the slices within a single grid). Moreover, since LGF stores both out-edge and in-edge slices, it supports bidirectional traversal and is well suited for the reverse plans required by WavePlan. In particular, subdividing blocks into smaller slices enables LGF to materialize RPQ results in a partitioned form, ensuring that even the potentially huge result sizes of all-pairs RPQs can be stored slice by slice. For example, Figure 4(b) shows the results of two RPQ atoms required for Q_2 in Figure 1. P'_0 and P'_1 correspond to the RPQ results of ab and c^*, respectively, where P'_0 contains two

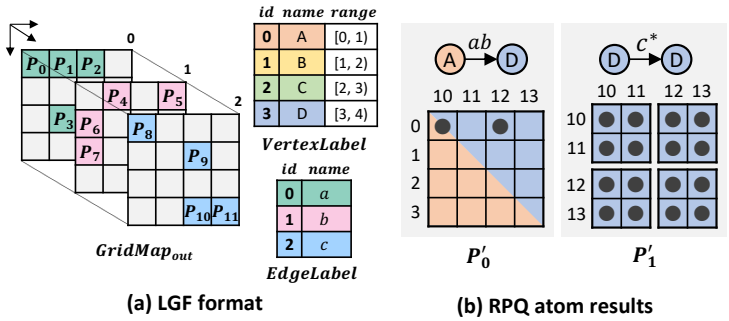


Fig. 4. Example of Labeled Grid Format (LGF).

Table 1. Example of edges stored in each slice.

Edge labels	Edges per slice
a	S_0: (v_0, v_1), (v_0, v_3), S_1: (v_2, v_5), S_2: (v_0, v_6), S_3: (v_7, v_5)
b	S_4: (v_1, v_4), S_5: (v_1, v_{10}), (v_3, v_{12}), S_6: (v_5, v_2), S_7: (v_6, v_1)
c	S_8: (v_2, v_3), (v_3, v_2), S_9: (v_4, v_7), S_{10}: (v_{10}, v_8), (v_{13}, v_9) S_{11}: (v_{10}, v_{11}), (v_{11}, v_{12}), (v_{12}, v_{13}), (v_{13}, v_{10})

edges $\{(v_0, v_{10}), (v_0, v_{12})\}$. If the threshold is limited to four edges per slice, block P'_1 is subdivided into four quadrants along the source and destination dimensions, resulting in four slices.

3 Challenges of GPU-Based CRPQ Processing

In this section, we elaborate on the challenges highlighted in the Introduction, based on the discussion in Section 2.

Challenge 1 (Incorrect answers caused by traversal path length limitations): Most GPU-based graph query processing methods [13, 46, 49, 61, 66, 72, 82, 84, 88] adopt DFS-based traversal strategies due to the limited capacity of GPU and shared memory. Some hybrid methods [46, 49, 66, 72, 84] partially incorporate BFS, but GPU memory constraints for storing all frontier vertices prevent full BFS traversal [72, 84], leaving them essentially DFS-based. In detail, as defined in Definition 2.1, evaluating an RPQ requires exploring paths starting from $x = v_i$ that satisfy the regular expression ρ. On GPUs, this exploration is typically performed in a depth-first manner along with automaton transitions. However, DFS tends to explore longer paths first, which leads to inefficiency by unnecessarily expanding paths even when shorter valid ones exist. For example, Q_1 in Figure 1 yields the RPQ result pair (v_0, v_9), obtained through the 4-hop path $(v_0 \rightarrow v_3 \rightarrow v_{12} \rightarrow v_{13} \rightarrow v_9)$. Nevertheless, a DFS-based method would first explore a longer 6-hop path such as $(v_0 \rightarrow v_1 \rightarrow v_{10} \rightarrow v_{11} \rightarrow v_{12} \rightarrow v_{13} \rightarrow v_9)$.

Moreover, as the hop length increases, the amount of data that must be maintained in GPU and shared memory during traversal (e.g., slices, DFS stacks) also grows, making it impractical to set a sufficiently large hop bound. Since the entire data graph often cannot fit into GPU memory, traversal is typically performed by partitioning the graph and loading only a subset [38, 46, 61, 66, 91]. In practice, this constraint limits the hop bound to only a few dozen. DFS-based traversal can easily exceed these bounds, leading to missed valid paths and ultimately incorrect results. As shown in the experiments of Section 8.2, BFS completed traversal within 5 hops, while the DFS-based hybrid approach explored up to 40 hops — the maximum range allowed by GPU and shared memory — but still failed to produce correct results.

Challenge 2 (Limited parallelism due to visited set maintenance): As discussed in Section 2.2, the automata-based approach maintains a visited set to avoid redundant exploration and ensure distinct vertex pairs. The state-of-the-art CPU-based method Ring-RPQ [7] also manages a visited set for each starting vertex using a bitmap. In this case, the minimum memory usage per starting vertex is defined as $\frac{|V|\times|Q|}{8}$ (in bytes), where $|V|$ is the number of vertices in the data graph and $|Q|$ is the number of states in the automaton. For the LDBC SNB dataset with scale factor 10 and RPQ Q4 ($|V| = 35.5\,\text{M}$, $|Q| = 5$) used in Section 8.2, the memory requirement per starting vertex is $(35.5\,\text{M} \times 5)/8 \approx 22.2\,\text{MB}$.

On GPUs, it is common to assign each starting vertex to a separate thread block to exploit parallelism [13, 61, 66]. Since GPUs are designed to activate thousands of thread blocks simultaneously, thousands of starting vertices should ideally be processed in parallel. However, GPU memory limitations make this difficult to achieve in practice. For example, in the same graph mentioned earlier, if 20 GB of GPU memory is allocated for the visited-set buffer, only $20\,\text{GB}/22.2\,\text{MB} \approx 922$ starting vertices can be processed concurrently. This limitation significantly reduces the effective utilization of available thread blocks and consequently leads to performance degradation.

Challenge 3 (Inability to evaluate CRPQs due to the explosion of RPQ results): CRPQs can be efficiently processed by first materializing the results of individual RPQ atoms and then combining them through the WCOJ-based CQ processing method proposed in [66]. However, since all-pairs RPQs enumerate every pair, their results can rapidly surpass GPU memory capacity. For example, in Figure 4(a), the block P_{11} pointed to by (3, 3, 2) in GridMap consists of slice S_{11}, which

contains four edges forming a cycle over the vertex set $\{v_{10}, v_{11}, v_{12}, v_{13}\}$. When evaluating c^* on this slice, all vertices in the cycle become mutually reachable. This result explosion also occurs on CPUs. Even state-of-the-art CPU-based DBMSs Umbra [58] and DuckDB [70], which support CRPQs, can experience drastic increases in main-memory usage or even query failures during the materialization of RPQ atoms. These systems follow the algebra-based approach and thus even the intermediate results must be materialized, leading to excessive memory consumption.

Recently, several studies have proposed methods for handling graph query results that exceed GPU memory capacity [39, 46, 65]. These approaches typically enumerate large results rapidly and simply spill them into main memory. In contrast, when the results need to be reused as input for subsequent operations (e.g., WCOJ), they must be converted into a GPU-aware graph format (e.g., LGF). In particular, the parallel execution of GPU threads generates RPQ results in an irregular order [46, 65], so all results must be fully enumerated before conversion into a graph format. However, converting on GPUs may exceed memory capacity, while converting on CPUs keeps the GPU idle until completion and reduces the benefits of acceleration.

4 GPU-Based Hop-Limited Level-Wise DFS

In this section, we introduce the hop-limited level-wise DFS method, designed for efficient exploration of arbitrary path lengths in RPQs. This method traverses the graph from a starting vertex in a fixed interval, referred to as a *static-hop*. It first explores all paths within the static-hop bound, and if additional extensions are required during traversal, the search range is progressively expanded by further static-hop intervals. Hop-limited level-wise DFS can be described in two phases: (1) a **base phase**, which performs the initial exploration up to the static-hop bound, and (2) an **expansion phase**, which repeatedly extends the search range by static-hop intervals as needed.

4.1 Base Phase

The base phase identifies the partitions (slices) to be explored within the static-hop bound using the automaton derived from the regular expression, and then performs traversal on the GPU. We construct an *RPQ traversal tree* that organizes candidate slices satisfying the regular expression into a tree structure, extending the concept proposed in [66]. Starting from the initial state of the automaton and following its transitions, a *candidate slice* is selected only if the following two conditions are satisfied:

- The transition's edge label matches the candidate slice's label.
- If a parent slice exists, it can connect to the candidate slice to form a path.

This connectivity condition is determined by checking the overlap of the source and destination vertex ID ranges (src-range, dst-range), which represent only actual vertices per slice and are precomputed during LGF construction. Each node in the RPQ traversal tree stores its slice together with the automaton state reached through the transition, allowing us to check if the node is in a final state.

Example: Figure 5(a) shows the RPQ traversal tree for Q_1. We assume the vertex ID of v_i to be i, and use the plan in Figure 2(a). From the initial state q_0, the outgoing transition labeled a selects slices S_0, S_1, S_2, and S_3 as the roots of the tree. For S_0, the dst-range [1, 4) is checked for overlap with the src-ranges of slices labeled b. Since the src-ranges of S_4 and S_5 are [1, 2) and [1, 4), respectively, these slices are considered candidates, while the others (e.g., S_6, S_7) are pruned. Assuming a static-hop of 3, the tree construction terminates at depth 2. The tree nodes at depth 0 reach automaton state q_1, while those at depths 1 and 2 reach state q_2. The path through slices $S_0 \rightarrow S_4 \rightarrow S_9$ (e.g., $v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_7$) matches abc and reaches the final state of the automaton, producing a valid RPQ result.

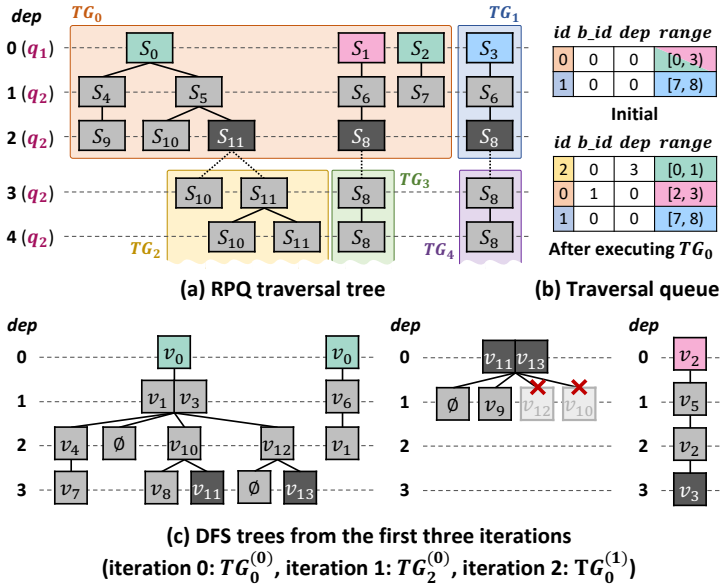


Fig. 5. Example of hop-limited level-wise DFS.

Since LGF is a 2D format partitioned by source and destination, root slices with the same block row ID can share starting vertices (e.g., both S_0 and S_2 include v_0). To enable traversal at the granularity of starting vertices, such subtrees are grouped into a *Traversal Group* (TG), which serves as the basic processing unit of traversal. For example, in Figure 5(a), two TGs are generated in the base phase, since slices S_0, S_1, and S_2 have block row ID 0, while S_3 has block row ID 2. TG IDs are assigned sequentially, and the TG with ID i is denoted TG_i.

After the slices that compose a TG are loaded into an *input buffer* in GPU memory, each TG is processed on the GPU in two steps: (1) *determining the set of starting vertices* and (2) *performing path exploration*. As we pointed out in Section 3, processing all starting vertices at once is infeasible in all-pairs RPQs due to GPU memory constraints. Therefore, in the first step, the starting vertices are divided into batches of fixed size, referred to as the *batch size*, determined by GPU occupancy (e.g., 4,096 concurrent starting vertices). In detail, the starting vertices for each batch are determined by performing a *k-way merge* [83] over the source vertex arrays of the TG's root slices. In the second step, path exploration is performed on the TG using the DFS method proposed in [66]. Traversal begins at the root slice with each starting vertex. Paths are then extended one hop along the edges in each slice, and the newly reached vertices are recursively explored through the child slices. Each batch in TG_i is assigned a sequential *batch ID* starting from 0, and the batch with ID j is denoted $TG_i^{(j)}$. A TG is fully processed once the starting vertices of all its root slices have been completely explored.

Example: We consider TG_0 in Figure 5(a). Assume that the eleven slices comprising TG_0 have been fully loaded into GPU memory. The source vertex sets of S_0, S_1, and S_2 are $\{v_0\}$, $\{v_2\}$, and $\{v_0\}$, respectively. When the batch size is set to 1, batch 0 selects $\{v_0\}$ as the starting vertex set. The left side of Figure 5(c) shows the DFS tree of $TG_0^{(0)}$, which depicts the exploration order of the path exploration process. Since S_1 does not contain any edges starting from v_0, it is excluded, and exploration begins from S_0 and S_2. From v_0 through S_0, vertices $\{v_1, v_3\}$ are reached, which then leads to exploration of S_4 and S_5. Following DFS order, exploration first proceeds through S_4, yielding the 3-hop path ($v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_7$) via S_9. Next, from S_5, traversal continues through S_{10} and S_{11}, reaching $\{v_8\}$ and $\{v_{11}, v_{13}\}$, respectively. Once the exploration of $TG_0^{(0)}$ is completed,

batch 1 explores paths starting from $\{v_2\}$ (the right side of Figure 5(c)). In this example, TG_0 is fully processed after two batches. $TG_2^{(0)}$ will be explained later.

4.2 Expansion Phase

Whenever each batch of a TG in the base phase completes and traversal can proceed beyond the static-hop boundary, the expansion phase is triggered. The expansion phase extends the RPQ traversal tree by the length of the static-hop from the leaf nodes of the parent TG at the static-hop boundary and continues the exploration. Each sub-tree extended from a single leaf node is defined as a new TG. Such a TG is referred to as an *expansion-TG*, whereas a TG generated in the base phase is called a *base-TG*. During the processing of an expansion-TG, the same expansion procedure can be recursively applied whenever additional extension is required. The base-TG determines its own set of starting vertices before traversal, whereas the expansion-TG inherits the set from its parent and executes only the second step (i.e., the path exploration step). At this step, traversal continues from the *checkpoint set* of each starting vertex, which contains the vertices reached at the static-hop boundary. We define the *checkpoint set* in Definition 4.1.

Definition 4.1. **Checkpoint set**: We define the *checkpoint set* for a start vertex s as the set of vertices reached at the static-hop boundary from the slices corresponding to leaf nodes of the parent TG. This set serves as the *exploration frontier* for s in the expansion phase.

Example: In Figures 5(a) and (c), the processing of $TG_0^{(0)}$ reaches two paths, $(v_0 \rightarrow v_1 \rightarrow v_{10} \rightarrow v_{11})$ and $(v_0 \rightarrow v_3 \rightarrow v_{12} \rightarrow v_{13})$, through S_{11}, which corresponds to a leaf node located at the static-hop boundary. In this case, we obtain the checkpoint set $\{v_{11}, v_{13}\}$ from the starting vertex v_0. The expansion phase is performed when candidate slices connected to the checkpoint set, serving as the frontier, are available. In detail, expansion at S_{11} occurs when the automaton contains a transition labeled c (e.g., a self-loop on q_2 with label c) and, at the same time, the vertex ID range of the checkpoint set $[11, 14)$ overlaps with the src-range of slices in the data graph (e.g., S_{10} and S_{11}). The sub-tree derived from S_{11} is grouped into a single expansion-TG, denoted TG_2. The middle part of Figure 5(c) shows the DFS tree of $TG_2^{(0)}$. Here, traversal resumes from the checkpoint set $\{v_{11}, v_{13}\}$, and an additional path $(v_{13} \rightarrow v_9)$ is discovered through S_{10}. This yields the RPQ result (v_0, v_9). Although traversal from S_{11} can reach v_{12} and v_{10}, these are paths already visited at depth 2 of $TG_0^{(0)}$ and are therefore skipped through the visited set management to be described in Section 5. Slices such as S_9 and S_{10} yield 3-hop paths (e.g., $v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_7$), but since no candidate slices are connected to these leaf nodes, no expansion phase is triggered.

Hop-limited level-wise DFS dynamically generates base-TGs and expansion-TGs during traversal, and their execution order is managed through a priority queue called the *traversal queue*. Each queue record stores the *TG ID*, *batch ID*, *depth*, and *start-vertex ID range*. TGs with greater depth have higher priority, and ties are broken by TG ID. When a TG is executed, new expansion-TGs are inserted into the queue, and the record of the original TG is updated to the next batch until all its starting vertices are processed. We call one such *dequeue-execution-enqueue* cycle an *iteration*, and traversal terminates when the queue becomes empty.

Example: Figure 5(b) shows how traversal progresses through the traversal queue. For brevity, each record is denoted as *id*, *b_id*, *dep*, and *range*. Records placed higher in the queue represent those with higher priority. In the initial state, the records of base-TGs TG_0 and TG_1 are inserted into the queue. The execution begins with $TG_0^{(0)}$, and once traversal with the starting vertex set $\{v_0\}$ is completed, the expansion phase generates TG_2. Here, the depth of TG_2 is 3, and its start-vertex ID range is inherited from the starting vertex set of $TG_0^{(0)}$, resulting in $[0, 1)$. Subsequently, the batch ID of TG_0 is incremented to 1, and since the processing of v_0 has been completed, its start

vertex range is updated to [2, 3). The entire execution consists of six iterations, following the order $TG_0^{(0)} \rightarrow TG_2^{(0)} \rightarrow TG_0^{(1)} \rightarrow TG_3^{(0)} \rightarrow TG_1^{(0)} \rightarrow TG_4^{(0)}$.

Hop-limited level-wise DFS differs from naive DFS by following depth-first exploration while collectively processing all paths within each static-hop boundary. This design effectively *incorporates the shortest-path-first property of BFS into arbitrary-path length queries*, reducing redundant exploration. For example, when naive DFS is executed in vertex ID order, the RPQ result (v_0, v_9) can only be reached via a **6-hop** path such as $(v_0 \rightarrow v_1 \rightarrow v_{10} \rightarrow v_{11} \rightarrow v_{12} \rightarrow v_{13} \rightarrow v_9)$. In contrast, as shown in Figure 5(c), $TG_0^{(0)}$ first discovers the path $(v_0 \rightarrow v_3 \rightarrow v_{12} \rightarrow v_{13})$, after which $TG_2^{(0)}$ extends it with $(v_{13} \rightarrow v_9)$, reaching the same result in only **4 hops**.

5 GPU-Based On-Demand Segment Pooling

In this section, we explain the on-demand segment pooling technique, designed for the efficient management of visited sets within GPU memory. During each iteration, when traversing the batches of a TG, it is necessary to maintain the visited sets for up to the maximum batch size of starting vertices in order to detect duplicate paths and prevent redundant exploration (e.g., v_{12} and v_{10} in the middle part of Figure 5(c)). Thanks to the RPQ traversal tree and the LGF format, the slices to be examined during traversal are predetermined, and the maximum size of the reachable vertex set for each slice is bounded by the *block-level width* (e.g., four in Figure 4(a)). Thus, rather than maintaining a global visited set for the entire graph, only the visited sets required for traversal are managed in units called *segments*.

A segment is a logical unit formed by partitioning a pre-allocated *segment buffer* in GPU memory into bitmap regions of block-level width. Each segment is assigned a sequential ID starting from 0. Free segments are managed in a *segment pool*, and each TG node (e.g., eleven TG nodes for TG_0 in Figure 5(a)) is assigned the required segment IDs from the pool at the beginning of the path exploration step. Once the traversal of the current TG completes and the segments are no longer used, they are returned to the pool. Segments are not limited to visited set management but may serve multiple purposes, classified into three principal types: (1) **Visited segments**, which record previously traversed paths to prevent redundant exploration, (2) **Checkpoint segments**, which preserve vertices reached at the static-hop boundary (i.e., the checkpoint set) so that expansion-TGs may continue traversal, and (3) **Bridge segments**, which maintain traversal continuity when a TG is too large to fit in GPU memory, allowing execution at the sub-TG level.

5.1 Visited Segment

Each segment is uniquely identified by a search context defined as (*starting vertex ID*, *automaton state*, *column ID*). Segments are assigned based on this key, and nodes with the same key share the same segment. Here, the column ID refers to the block column ID of the block to which a slice belongs. For example, in Figure 4, both S_9 and S_{10} have column ID 2. This mapping is managed through the *segment pooling table*, and if no segment exists for a given key, a new one is allocated from the segment pool. For single-source RPQs, each TG node is assigned exactly one visited segment, whereas for all-pairs RPQs, each node is assigned a number of visited segments equal to the batch size.

Example: Figure 6 shows how $TG_0^{(0)}$ in Figure 5 acquires visited segments using on-demand segment pooling. The starting vertex set of $TG_0^{(0)}$ is $\{v_0\}$. Among the sub-trees of $TG_0^{(0)}$, the one rooted at S_1 is omitted because it does not contain v_0. The rounded rectangles indicate the segment IDs mapped as visited segments to each node. Figure 6(b) shows the segment buffer and the segment pooling table, where the buffer holds several segments represented as 4-bit bitmaps. For brevity, the segment ID is denoted as *id*, and the key as $(v\_id, state, col\_id)$. Among the eight TG nodes, S_9

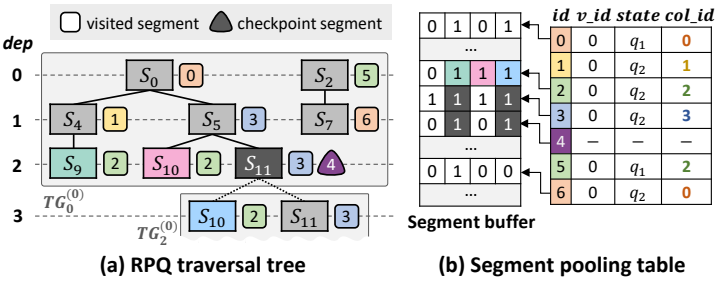


Fig. 6. Example of on-demand segment pooling method.

and S_{10} have the same segment ID, as do S_5 and S_{11} (e.g., 2 and 3, respectively). During the path exploration step, traversal reaches $\{v_7\}$ from S_9 and $\{v_8\}$ from S_{10}. This sets the second and third bits of the bitmap for segment ID 2. In contrast, S_0 and S_7 share the same column ID (e.g., 0) but correspond to different automaton states (e.g., q_1 and q_2), and thus map to different segment IDs.

A visited segment is released once the traversal of a TG and all of its derived expansion-TGs is complete. For example, in Figure 6, after $TG_0^{(0)}$ completes, the visited segments are still retained for the exploration of the derived $TG_2^{(0)}$. When $TG_2^{(0)}$ completes, the related visited segments (e.g., six in total) are released. Since the traversal queue prioritizes deeper TGs, visited segments are returned quickly instead of being retained for extended periods.

5.2 Checkpoint Segment

A checkpoint segment stores the checkpoint set in bitmap form, enabling exploration to continue within an expansion-TG. Each leaf node at the static-hop boundary allocates checkpoint segments. In a single-source RPQ, each leaf node is assigned exactly one checkpoint segment, whereas in an all-pairs RPQ, each leaf is assigned a number of checkpoint segments equal to the batch size. A checkpoint segment is released once its expansion-TG completes.

Example: In Figure 6(a), the leaf node S_{11} is assigned a checkpoint segment with ID 4. During traversal, the second and fourth bits of this segment are set to represent the checkpoint set $\{v_{11}, v_{13}\}$. This set then serves as the traversal frontier for the expansion-TG, TG_2. When TG_2 completes, the checkpoint segment (ID = 4) is released. If there are no candidate slices for the next hop, no checkpoint segment is allocated (e.g., S_9 and S_{10}).

5.3 Bridge Segment

When the data graph is large or the exploration range broad, processing a single TG in one execution may be difficult. For example, the slices of the TG may not all fit into the input buffer, or even when all segments are utilized, execution may still be infeasible. To address this issue, cuRPQ applies *sub-TG partitioning*, which divides a TG into smaller units called *sub-TGs*. A sub-TG is defined along a *root-leaf tree path* of the TG, where the consecutive nodes along the tree path to each leaf are grouped into a sub-TG. When multiple leaves share ancestor nodes, those nodes are duplicated across the corresponding sub-TGs. During this process, the input buffer usage and the number of segments are cumulatively estimated along each tree path. If execution becomes infeasible, the tree path explored so far is finalized as a sub-TG, after which the procedure proceeds with the remaining nodes. For example, Figure 7(a) shows TG_0 partitioned into four sub-TGs, $sub\text{-}TG_{0..3}$. In this case, the tree paths $\{S_0, S_4, S_9\}$ and $\{S_0, S_5, S_{10}\}$ belong to $sub\text{-}TG_0$, while the path $\{S_0, S_5, S_{11}\}$ forms a separate $sub\text{-}TG_1$.

When a TG is divided into multiple sub-TGs, **two major issues** arise. **First,** *because the same starting vertex may appear in multiple sub-TGs, its visited segments must be retained until all relevant sub-TGs have completed.* To avoid this, we extend the priority policy of the traversal queue so that

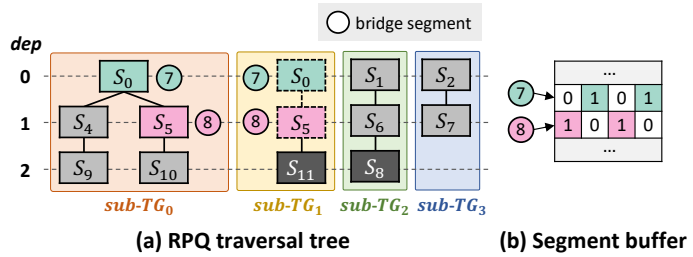


Fig. 7. Example of sub-TG partitioning.

sub-TGs are executed in ascending order of their start-vertex ID ranges, ensuring that each starting vertex is processed sequentially. We release the visited segments only after the vertex has been fully processed across all sub-TGs. For example, in Figure 7(a), the start-vertex ID ranges of $sub\text{-}TG_{0..3}$ are [0, 1), [0, 1), [2, 3), and [0, 1), respectively. Thus, $sub\text{-}TG_0$, $sub\text{-}TG_1$, and $sub\text{-}TG_3$ are processed in order, after which the visited segments for v_0 are released, followed by the execution of $sub\text{-}TG_2$.

Second, *when consecutive sub-TGs share common nodes at their boundary, path exploration may be prematurely blocked, since those nodes could have already been marked as visited in an earlier sub-TG.* We refer to these common nodes as the *cut set*. For example, the cut set between $sub\text{-}TG_0$ and $sub\text{-}TG_1$ is $\{S_0, S_5\}$. If $\{v_1, v_3\}$ is first reached through S_0 in $sub\text{-}TG_0$ and marked as visited, then in $sub\text{-}TG_1$, the path through slices $S_0 \rightarrow S_5 \rightarrow S_{11}$ can no longer be explored.

We address this issue by storing the reachable vertices of the cut-set nodes in bridge segments and passing them to the subsequent sub-TG. This ensures that when the same slice is revisited, exploration proceeds only through the permitted vertices recorded in the bridge segment, regardless of the visited segment. For example, in Figure 7, when traversing $sub\text{-}TG_0$, vertices $\{v_1, v_3\}$ from S_0 and $\{v_{10}, v_{12}\}$ from S_5 are recorded in segments 7 and 8, respectively. When $sub\text{-}TG_1$ is executed, the bridge segments are passed from $sub\text{-}TG_0$, and traversal continues by following only the edges leading to vertices whose bits are set, regardless of the visited set. In a single-source RPQ, each cut-set node is assigned one bridge segment, whereas in an all-pairs RPQ, it is assigned a number of bridge segments equal to the batch size. Once the subsequent sub-TG completes, the corresponding bridge segments are released.

6 Concurrent Exploration-Materialization

In this section, we explain the concurrent exploration-materialization strategy in cuRPQ, designed to efficiently manage large RPQ results. Section 6.1 introduces the *batch-incremental materialization* method, which enables concurrent GPU-side exploration and CPU-side materialization. Section 6.2 describes how this method extends to CRPQ processing and execution plan strategies.

6.1 Batch-Incremental Materialization

To store large RPQ results efficiently on the GPU and enable their reuse in subsequent operations (e.g., CRPQ or RPQ), cuRPQ materializes the results in the LGF format. The result of a single RPQ forms a grid, where each node in the RPQ traversal tree corresponds to a block. A TG corresponds to the blocks within a single row of this grid, all of which share the same block row ID. For example, in Figure 5(a), S_4 of TG_0 corresponds to coordinate $(0, 1)$, defined by the block row ID of its root node and the block column ID of the slice, while both S_9 and S_{10} map to $(0, 2)$. A straightforward approach would be to allocate a GPU memory buffer for each block and store the corresponding RPQ results directly. However, block-level result sizes are difficult to predict in advance. This leads to excessive space allocation, and large outputs can easily exceed GPU memory capacity, making execution infeasible.

cuRPQ adopts an approach where results from all blocks are aggregated into a *unified RPQ results buffer* (UR buffer). Following the technique proposed in [65], the UR buffer is divided into small chunks, and each thread writes results sequentially into its assigned chunk. When a chunk becomes full, a new chunk is allocated. If no more chunks are available, the kernel suspends execution, transfers the accumulated results from the UR buffer to main memory, and then resumes kernel execution. This strategy keeps GPU memory usage stable. However, the drawback remains that RPQ results must still be materialized in bulk, leading to high reconstruction overhead. Furthermore, ensuring that each slice remains below a threshold θ (e.g., block P'_1 in Figure 4(b)) requires repeated recursive quadrant partitioning, which often becomes a major bottleneck.

We explain a *Batch-Incremental Materialization* (BIM) method that allows the CPU to progressively materialize RPQ results during query execution. The key idea is to accumulate results block by block during each UR buffer transfer and to progressively materialize completed slices on the CPU during query execution. Since both TGs and sub-TGs are processed in batches ordered by starting vertex IDs, the results within each block are also filled in this order. Thus, once the exploration for a certain vertex range has finished, the corresponding slices can be safely materialized. This BIM process consists of three steps.

- **Step 1:** RPQ results accumulated in the UR buffer are transferred from the GPU to main memory via device-to-host (D2H) transfer.
- **Step 2:** The CPU scans the transferred results, determines the block to which each result belongs, and stores them in the corresponding *temporary slice buffer* (T_i). Each T_i is an intermediate buffer with a capacity of θ, used to accumulate results before slice materialization. Initially, one T_i is allocated per block, but when a buffer becomes full, the slice is partitioned into quadrants and up to four new T_i buffers are allocated.
- **Step 3:** When the exploration of a slice's starting vertex range completes, its corresponding T_i is materialized as a slice.

Figure 8(a) shows an example of BIM. At this point, results are being produced for three blocks, and four temporary buffers $T_{0..3}$ are maintained in main memory. Assuming the block width and height are n, T_0 covers the range $[0, n/2) \times [0, n/2)$ and T_1 covers $[0, n/2) \times [n/2, n)$. By contrast, T_2 and T_3 have not been partitioned yet and still cover the entire range $[0, n) \times [0, n)$. During Step 2, if T_0 becomes full, it is replaced by four new buffers T_4-T_7 created through quadrant partitioning. The results stored in T_0 are redistributed among these new buffers based on their ranges, and T_0 is released. During Step 3, once the TG (or its sub-TGs) associated with block B_0 has completed exploration of the starting vertex range $[0, n/4)$, the corresponding buffers T_4 and T_5 are materialized as slices.

This approach can be extended to support asynchronous CPU-side BIM, which overlaps with the RPQ kernel through GPU streams. Figure 8(b) shows a GPU-CPU pipeline that overlaps query

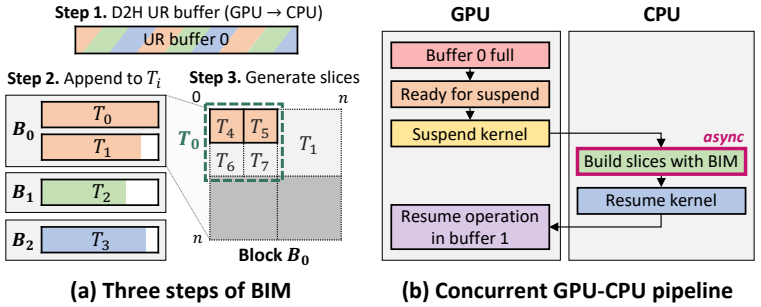


Fig. 8. Example of concurrent exploration-materialization.

execution with materialization by alternating between two UR buffers. When UR buffer 0 becomes full, the kernel pauses briefly and signals the CPU to perform the BIM step asynchronously, starting with a D2H transfer (Step 1). Meanwhile, the GPU resumes exploration using UR buffer 1. By alternating between the two buffers, query execution and materialization are overlapped.

6.2 Processing CRPQs with Execution Plans

After materializing the results of RPQ atoms, we process a CRPQ by combining them using the WCOJ-based CQ method proposed in [66]. However, depending on the matching order in WCOJ [46, 66, 91], the exploration direction of an RPQ atom may differ from the direction required in the join. For example, even if exploration is performed and materialized in the out-edge direction (i.e., out-edge slice), WCOJ may require an in-edge slice instead. To address this, the CRPQ plan uses the matching order to determine the required direction for each RPQ atom in advance and materializes results only in that direction. If the opposite direction is required, we perform a *slice transpose*, denoting the corresponding automaton A as A^T in the plan.

Figure 9(a) shows the CRPQ plan for Q_2 in Figure 1. Each edge is annotated with the automaton of its corresponding RPQ atom, indicating that their results will be joined using WCOJ. Suppose the automata for $x \xrightarrow{ab} y$ and $x \xrightarrow{c^*} y$ are A_{ab} and A_{c^*}, respectively, and assume the matching order $u_2 \prec u_3 \prec u_4$. In this case, A_{c^*} requires only out-edge slices, whereas A_{ab} requires both out-edge and in-edge slices for WCOJ. The key point is that while A_{ab} requires results in both directions, exploration is done only once, with materialization producing both out-edge and in-edge slices.

In addition, cuRPQ can support diverse execution plan strategies such as WavePlan [87]. For example, the *reverse* plan (e.g., A_1 in Figure 3) can be handled by using in-edge slices in the LGF format ($GridMap_{in}$). The *loop-cache* plan (e.g., A_2) leverages cuRPQ's materialization strategy, where partial query results (e.g., A'_2) are first materialized as slices and then reused in subsequent RPQ exploration to extend paths. In contrast, the *start-in-the-middle* plan (e.g., A_3 and A_4) begins exploration from the middle of the path, so result pairs cannot be confirmed in order of start vertices as in forward exploration. Such plans must first enumerate all result pairs before materialization, and therefore BIM cannot be applied directly. To execute these plans under limited GPU memory, cuRPQ preserves the start-in-the-middle strategy while leveraging slice transpose. In this approach, one sub-RPQ result is materialized in a single direction and then transposed into the opposite direction for joining. For example, Figure 9(b) shows cuRPQ's plan for A_3, where the out-edge paths satisfying bc^* (e.g., A'_5) are first computed and then transposed into in-edge slices to join with a (e.g., A_5).

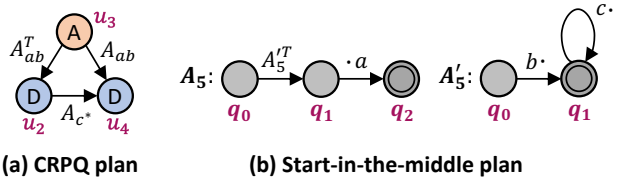


Fig. 9. Example of execution plan strategies.

7 Architecture of cuRPQ

Figure 10 provides an overview of the cuRPQ architecture, which consists of three main layers: the *query interpretation layer*, the *execution engine layer*, and the *data access layer*. Given an RPQ or CRPQ as input, the query interpretation layer parses the query and translates it into a corresponding automata-based execution plan. The execution engine layer then performs RPQ traversal based on this automaton. Specifically, the traversal group manager constructs the RPQ traversal tree and

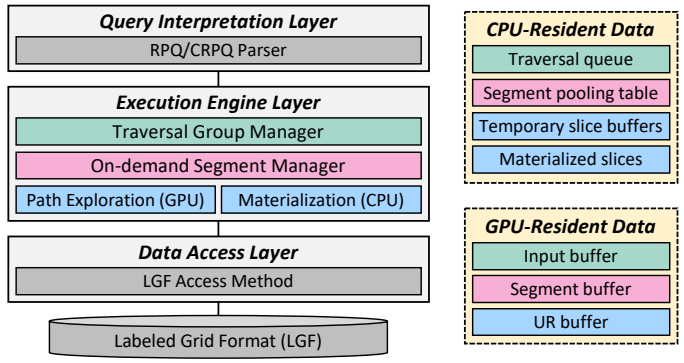


Fig. 10. Architecture of cuRPQ and its core metadata structures.

manages TGs via the traversal queue (Section 4). For each TG, the on-demand segment manager allocates the required segments during traversal (Section 5). Path exploration is executed on the GPU, while result materialization is performed concurrently on the CPU (Section 6). At the data access layer, graph data stored on disk in the LGF format is accessed on demand during execution.

Algorithm 1 presents the overall workflow of the execution engine layer. cuRPQ processes each RPQ atom in the execution plan sequentially (Line 1). For each atom, it first executes the base phase (Lines 2-3). In each iteration, the execution engine performs traversal for a TG selected from the traversal queue (Lines 4-14) and extends the TG via the expansion phase when necessary to enable further traversal (Lines 12-13). During the execution of each TG, the engine loads the required slices, allocates the necessary segments, and launches the path exploration kernel on the GPU (Lines 6-8). When the UR buffer reaches its capacity, the execution engine asynchronously materializes slices in the UR buffer on the CPU and then resumes GPU kernel execution, thereby overlapping exploration and materialization (Lines 9-11). For CRPQs, once all RPQ atoms have been processed, the system performs WCOJ-based conjunctive query processing over the materialized slices (Lines 15-16).

The path exploration kernel utilizes all available GPU thread blocks. Following widely adopted strategies in GPU-based graph analytics [13, 61, 66], each starting vertex in a TG is mapped to a separate thread block to exploit parallelism. Within each thread block, threads process the adjacency list of the slice corresponding to the current TG node during hop-limited level-wise DFS. Destination vertices are accessed in a coalesced manner, enabling efficient warp-level memory access. Due to hop-limited traversal, each TG forms a tree structure that is small enough to reside in shared memory, thereby reducing random global memory accesses during traversal.

Figure 10 also shows the core metadata structures maintained during execution. The traversal queue prioritizes deeper TGs, which prevents the queue from continuously growing with shallower TGs and keeps the queue size small. For the LDBC SNB dataset with scale factor 10 and RPQ Q4 used in Section 8.2, the traversal queue maintains at most 34 entries, resulting in only a few kilobytes of overhead. The segment pooling table size is bounded by the number of pre-allocated segments and remains within a few megabytes. In addition, the input buffer, segment buffer, and UR buffer are allocated with fixed sizes on the GPU, avoiding additional memory allocation during execution.

8 Performance Evaluation

8.1 Experimental Setup

RPQ datasets and queries: We adopt datasets and queries widely used in prior studies on RPQs in streaming graph environments [35, 62, 63]. The ten RPQ queries used in our evaluation are

Algorithm 1: Execution Workflow in cuRPQ

Input: RPQ/CRPQ execution plan P

```
foreach RPQ atom a in P do
    Generate base-TGs for a;
    Enqueue base-TGs into traversal queue;
    while traversal queue is not empty do
        tg ← Dequeue a TG from traversal queue;
        Load slices of tg into GPU input buffer;
        Allocate required segments for tg;
        Launch the path exploration kernel for tg;
        while UR buffer is full do
            Asynchronously materialize RPQ results;
            Resume the path exploration kernel for tg;
        if expansion phase is triggered for tg then
            Enqueue expansion-TGs into traversal queue;
        Release allocated segments of tg;
if P is CRPQ then
    Execute a WCOJ-based CQ kernel on materialized slices;
```

summarized in Table 2. Among them, nine are recursive queries that frequently appear in real-world applications [15, 16], while Q4 is a non-recursive query that we slightly modified for use as a baseline. These recursive query patterns have been reported to cover more than 99% of all recursive RPQs observed in Wikidata query logs [15]. For the data graphs, we use both synthetic and real-world datasets. As the synthetic dataset, we use the **LDBC SNB** [73], which simulates real-world interactions in social networks. This dataset contains 12 types of vertex labels and 15 types of edge labels. We use two *scale factors* (SF): SF=1 (4.0 M vertices and 22.4 M edges) and SF=10 (35.5 M vertices and 219.4 M edges). Table 3 shows the number of results for each query. As the LDBC SNB dataset contains only two recursive edges (e.g., *ReplyOf* and *Knows*), some recursive queries such as Q8, Q9, and Q10 cannot be meaningfully represented; therefore, their result counts are omitted from Table 3, as in [35, 62].

As the real-world dataset, we use **StackOverflow** [48], which contains 3 types of vertex labels and 3 types of edge labels, spanning 7 years and 7 months. Based on timestamps, we divide it into two spans: Span=6M (22.1 K vertices and 433.9 K edges) and Span=1Y (46.6 K vertices and 1.3 M edges). All queries return the number of result pairs satisfying the given regular expression.

Table 2. RPQ queries used in the experiments.

Notation	Query	Notation	Query
Q1	a^*	Q6	ab^*c
Q2	$a?b^*$	Q7	$(a_1 + a_2 + ... + a_k)b^*$
Q3	ab^*	Q8	a^*b^*
Q4	$abcd$	Q9	ab^*c^*
Q5	abc^*	Q10	$(a_1 + a_2 + ... + a_k)^*$

CRPQ datasets and queries: We evaluate CRPQ performance using the five queries (CQ1-CQ5) shown in Figure 11. The data graph is the same LDBC SNB dataset used for the RPQ experiments, and

Table 3. Number of result pairs per query.

SF	Q1	Q2	Q3	Q4	Q5	Q6	Q7
1	105.6 M	1.2 B	1.0 B	0.5 M	4.2 B	137.1 M	39.4 B
10	4.9 B	53.2 B	48.3 B	3.6 M	72.2 B	930.7 M	2.4 T

we consider two scale factors: SF=1 and SF=10. The queries are derived from the LDBC microbenchmark *Large-scale Complex Subgraph Query Benchmark* [54], which focuses on conjunctive queries. We extend some edges by incorporating transitive closure, transforming the original conjunctive queries into CRPQs. Vertex labels are distinguished by colors, and edges are annotated with the corresponding regular expressions. CQ4 and CQ5 include a pair of vertices represented by dashed lines, which indicate a filter condition that both vertices must map to distinct vertices in the data graph.

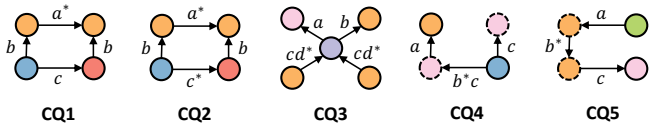


Fig. 11. CRPQ queries used in the experiments.

Systems compared: We compare cuRPQ with state-of-the-art methods. First, we compare with DuckDB [70] and Umbra [58], which are CPU-based relational DBMSs following an algebra-based approach. We also compare with Ring-RPQ [7], a CPU-based system that follows an automata-based approach. Although Ring-RPQ supports *reverse* execution plan (e.g., A_1 in Figure 3), it does not support parallel execution. Therefore, we extend its implementation so that each CPU core can execute independently from different starting vertices. Table 4 summarizes the systems compared in our experiments.

Meanwhile, several well-known graph processing systems that support RPQ processing are excluded from this comparison. For example, Virtuoso [27] and Blazegraph [74] (CPU-based RDF store DBMSs), as well as TigerGraph [24] (a CPU-based graph DBMS), are excluded because all-pairs RPQ evaluation leads to exponential path growth and prohibitively long execution times. In addition, Kùzu [29] (a CPU-based graph DBMS), HeavyDB [40] (a GPU-based relational DBMS), and RAPIDS [18] (a GPU-based relational library) do not support transitive closure [56]. We provide a more detailed comparison of these systems in Section 8.2, including their performance on small-batch RPQ.

Table 4. Summary of characteristics of compared systems.

Approach	Systems	CPU/ GPU	Support CRPQ	In-memory/ Disk-based
Algebra-based	DuckDB	CPU	O	Disk-based
	Umbra	CPU	O	Disk-based
Automata-based	Ring-RPQ	CPU	X	In-memory
	cuRPQ	GPU	O	Disk-based

HW/SW environment: We conduct all experiments on a dual-socket server equipped with two AMD EPYC 7302 processors (16 cores each), featuring a hierarchical cache architecture with a total of 1 MiB L1d, 16 MiB L2, and 256 MiB L3 cache. The system is configured with 1 TB of DDR4 memory operating at 2933 MT/s (DDR4-3200 DIMMs), four 6.4 TB PCIe SSDs (RAID 0), and one NVIDIA A100 GPU with 80 GB of memory. For CPU-based methods, all CPU cores and memory

are utilized without restrictions, and their default settings are applied. For cuRPQ, unless otherwise specified, we allocate 24 GB as GPU buffer memory, with 2 GB for the input buffer, 20 GB for the segment buffer, and 2 GB for the UR buffer. The operating system is Ubuntu 20.04.02, with g++ 7.5.0 as the compiler, and CUDA Toolkit 11.6 for kernel compilation. The versions of the systems compared in the experiments are DuckDB 1.2.2 and Umbra 30b000783. To ensure fair comparisons, we evaluate the performance of all methods with memory and OS caches cleared before each execution (i.e., cold start). The only exception is Ring-RPQ, which operates in an in-memory fashion after loading the entire graph into main memory. Accordingly, graph loading time from SSDs to main memory is excluded, and only query execution time is measured.

8.2 Comparison of RPQ Performance

Figure 12 shows the query execution times of DuckDB, Umbra, Ring-RPQ, and cuRPQ on the LDBC SNB and StackOverflow datasets.

Comparison with algebra-based systems: For the LDBC SNB dataset, both DuckDB and Umbra fail to complete Q7 at SF=1, and they fail to process all queries except for the non-recursive query Q4 at SF=10. DuckDB encounters O.O.M. errors for Q5 and Q7 at SF=1 due to the large number of vertices to be explored, while the remaining queries terminate with T.O. Umbra shows overall better performance than DuckDB, but execution still aborts due to O.O.M. This indicates a limitation of algebra-based approaches, where the materialization of intermediate results by the α-operator can cause O.O.M. or excessive computational overhead leading to T.O. A similar limitation is also observed on the StackOverflow dataset. Interestingly, DuckDB fails with timeouts for Q3 and Q4 when Span=6M, but it completes without timeouts on the larger dataset (Span=1Y) due to differences in the execution plan. In contrast, cuRPQ successfully processes all queries. cuRPQ outperforms DuckDB by up to 4,945X (Q6 at SF=1), and Umbra by up to 115X (Q6 at SF=1).

Comparison with automata-based systems: Ring-RPQ fails to return correct results for specific queries such as Q4, Q6, and Q7 due to implementation issues. For evaluation, we allow errors within a 3% margin, where the *error rate* represents the proportion of result pairs that a system fails to find with respect to the ground truth. If the error rate exceeds this threshold, we denote the result as W.A. Ring-RPQ generally performs slower than Umbra but faster than DuckDB, except for Q5 at SF=1. Automata-based approaches can propagate results directly through state transitions in the automaton, thus enabling pipelined execution. Thanks to this property, Ring-RPQ can process queries without O.O.M. for Span=1Y, in contrast to algebra-based systems. However, at SF=10, the number of paths grows exponentially, leading all queries to terminate with T.O. or W.A. In comparison, cuRPQ outperforms Ring-RPQ by up to 269X (Q7 at SF=1).

Performance overhead of arbitrary path lengths in RPQ: Table 5 compares cuRPQ's Hop-limited Level-wise DFS (denoted as *HL-DFS*) with a method that handles arbitrary paths using DFS with a sufficiently large static-hop limit (denoted as *Naive-DFS*). When we vary the static-hop size, we measure the maximum hop depth reached during exploration. With a static-hop size of 2, HL-DFS explores up to 6 hops and finds all results. In contrast, even when Naive-DFS increases the static-hop size up to 40 — the maximum allowed in GPU and shared memory — it still yields an error rate of 7.1%. This is because DFS explores paths deeply, often leading to exploration of unnecessarily long paths. HL-DFS performs DFS within each TG, so expansion-TGs are still needed for long paths, though larger static-hop sizes reduce the number of required expansions (e.g., three expansions at static-hop=2 and two at 40). Figure 13(a) presents the query execution times of both methods. Both Naive-DFS and HL-DFS show increased execution time as the static-hop size grows due to their DFS nature, but HL-DFS can maintain efficiency with a reasonably small static-hop size. In our experiments, we set the static-hop size of cuRPQ to 5.

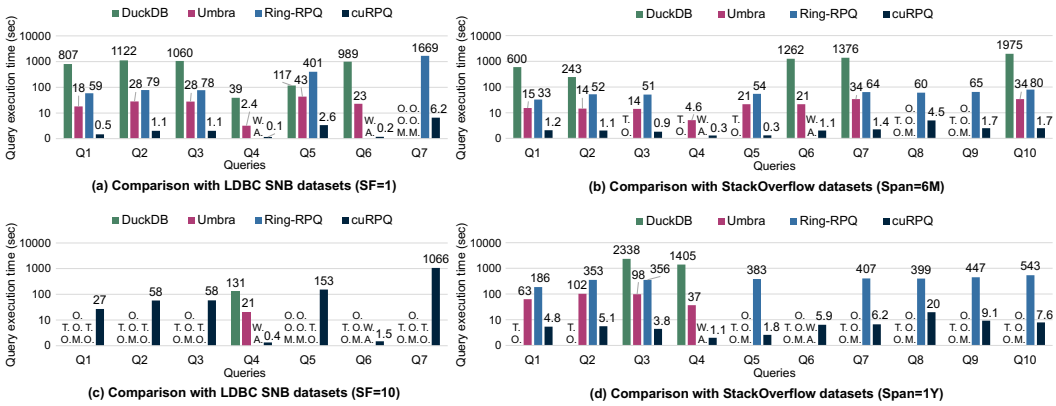


Fig. 12. Comparison of RPQ execution times (T.O.: elapsed time exceeds 1 hour, O.O.M.: out of memory, W.A.: wrong answer).

Table 5. Maximum hop counts with varying static-hop size.

Search	Maximum hops per static-hop size					Error rate range
	2	5	10	20	40	
Naive-DFS	≤ 2	≤ 5	≤ 10	≤ 20	≤ 40	7.1 % ~ 87.9 %
HL-DFS	≤ 6	≤ 15	≤ 20	≤ 40	≤ 80	0 %

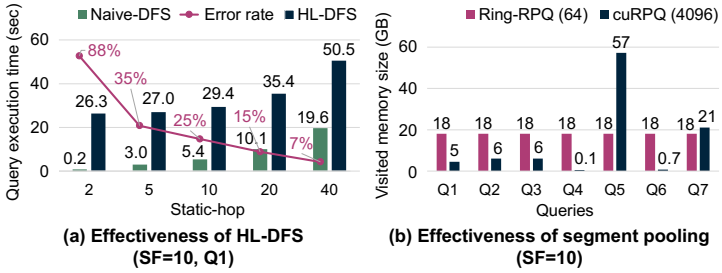


Fig. 13. Impact of HL-DFS and segment pooling.

Memory overhead from visited set maintenance: Figure 13(b) shows the maximum memory consumption of the visited set for two automata-based systems, Ring-RPQ and cuRPQ. Ring-RPQ processes up to 64 starting vertices simultaneously by leveraging 64 CPU virtual cores, whereas cuRPQ sets the batch size to 4,096 to exploit thousands of GPU cores. Ring-RPQ implements the visited set using a wavelet tree data structure that supports simultaneous state transitions, so its memory usage is theoretically somewhat larger but remains constant. In contrast, cuRPQ manages only the segments relevant to the query. Even when processing 4,096 starting vertices simultaneously — 64 times more than Ring-RPQ — it achieves significantly lower memory usage (180X lower for Q4). For Q5 and Q7, the memory demand approaches the segment buffer limit (20 GB), but buffer consumption is effectively controlled within 20 GB through sub-TG partitioning with bridge segments.

Comparison with small-batch RPQ: Figure 14 shows the query execution times of various systems that support RPQ when the number of starting vertices is 1, 64, and 128. To provide a more precise comparison, we conducted the evaluation on a smaller dataset with SF=0.1 (0.4M vertices and 2.1M edges). The timeout threshold is set to 5 minutes. The system versions used for

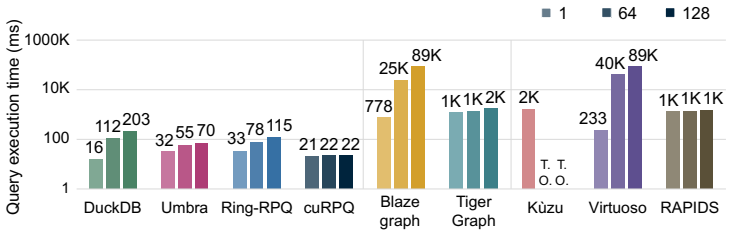


Fig. 14. Comparison of query execution times on small-batch RPQ (LDBC SNB, SF=0.1, Q1).

comparison are Blazegraph 2.1.6, TigerGraph Enterprise Edition 3.10.4, Kùzu 0.4.2, Virtuoso 7.2.15, and RAPIDS 24.12. For Kùzu and RAPIDS, which do not support the transitive closure, we limited the maximum hop length to 4, which may cause some paths (e.g., those reachable only through five hops) to be missed. Virtuoso supports transitive closure but lacks duplicate detection, leading to excessive redundant exploration and ultimately T.O. Thus, we limited Virtuoso to a maximum hop length of 4. HeavyDB does not support the α-operator or UNION operator, and is excluded from the comparison.

Systems other than the four main baselines considered in this paper show performance between 11X slower (Virtuoso) up to 4,045X slower (Blazegraph and Virtuoso) compared to cuRPQ. Blazegraph is known to internally optimize transitive closure operations, resulting in relatively faster performance for single starting vertices, but its performance degrades sharply as the number of starting vertices increases. RAPIDS efficiently utilizes the GPU, showing little to no performance degradation as the number of starting vertices increases. However, it can only process datasets that fit into GPU memory, and O.O.M. occurs for all-pairs RPQ. Although cuRPQ shows lower performance than DuckDB for single-source RPQ because only one thread block is used, it achieves substantial performance improvements as the number of starting vertices increases and the workload approaches all-pairs RPQ by exploiting massive GPU parallelism (9.2X at 128 vertices).

8.3 Comparison of CRPQ Performance

Figure 15 shows the query execution times of CRPQ queries on DuckDB, Umbra, and cuRPQ. CQ1 incorporates Q1 as an RPQ atom. Since algebra-based systems DuckDB and Umbra inevitably materialize intermediate results even for standalone RPQs, the additional cost of executing CQ1 compared to Q1 is negligible (e.g., 18 sec and 19 sec with Umbra). In contrast, cuRPQ incurs a modest overhead due to BIM (0.5 sec and 2.2 sec), yet it still outperforms DuckDB by up to 420X (CQ1 at SF=1) and Umbra by up to 23.2X (CQ5 at SF=1). The same trend is observed at SF=10. DuckDB and Umbra fail to execute CQ1, CQ2, and CQ5 due to T.O. and O.O.M., respectively, when performing transitive closure. cuRPQ successfully processes all queries and outperforms DuckDB by 20.7X and Umbra by 5.9X for CQ4.

RPQ result explosion for CRPQ: Figure 16(a) shows the peak main memory consumption during CRPQ execution. At SF=10, DuckDB encounters T.O. for CQ1, CQ2, and CQ5, but we report the peak memory usage observed within the 2-hour time limit. Umbra exhausts the entire 1 TB of memory on these three queries. In contrast, cuRPQ consumes up to 12.4X less memory than DuckDB for CQ5 and at least 21.3X less memory than Umbra for CQ1 and CQ5. For CQ3 and CQ4, cuRPQ uses slightly more memory, as it must maintain block-level temporary slice buffers that have not yet been materialized. However, cuRPQ immediately releases completed temporary slice buffers through BIM and keeps only the compressed graph format in main memory. Thus, cuRPQ effectively controls main memory usage within 64 GB even for queries such as CQ1, CQ2, and CQ5, where RPQ result explosion is particularly severe.

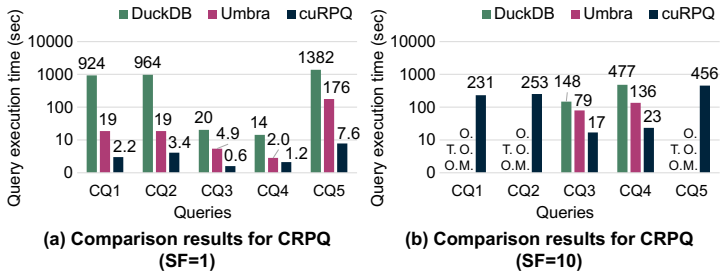


Fig. 15. Comparison of CRPQ execution times.

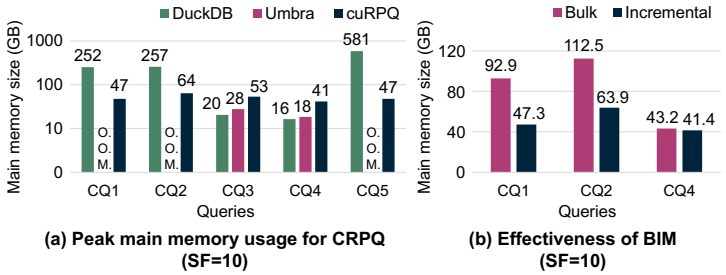


Fig. 16. Intermediate data sizes in CRPQ.

Figure 16(b) shows the peak memory consumption when results are materialized in bulk without BIM. Without BIM, all RPQ results from the UR buffer must be retained in main memory in enumerated form, causing memory usage to scale with the output size of the RPQ atoms in CRPQ queries. With BIM, memory consumption is reduced by up to 2.0X for CQ1.

8.4 GPU Performance and Parallelism Analysis

Table 6 reports the **GPU utilization metrics** of cuRPQ. The number of active warps per scheduler ranges from approximately 38-46%, while the number of active threads per warp varies between 67-75%. Given the inherently irregular control flow of graph traversal workloads, these values fall within a reasonable range [85]. Meanwhile, DRAM throughput remains relatively low, whereas the L2 cache hit rate consistently stays around 90%. This indicates that RPQ evaluation in cuRPQ is largely compute-dominant rather than memory-bandwidth-bound, making it well suited for GPU acceleration.

Table 7 summarizes the characteristics of **query-level parallelism** in cuRPQ. Depending on the query, the number of generated TGs exceeds one thousand, enabling effective coarse-grained parallel execution at the TG level. The TG depth reflects the depth of the TG hierarchy formed by successive expansions during the hop-limited level-wise DFS. As TG depth increases, the maximum reachable hop length is progressively extended according to the static-hop configuration; for example, Q8 at Span=1Y reaches up to $6 \times 5 = 30$ hops. Across queries, the fan-out in base-TGs and the TG depth exhibit diverse patterns. This diversity enables cuRPQ to accommodate both wide and deep traversal behaviors, effectively covering the arbitrary path lengths inherent in RPQ queries.

Table 8 shows the **breakdown of BIM execution time**. The overlap ratio measures the degree to which GPU-based exploration and CPU-side materialization are executed concurrently, normalized by the maximum possible overlap [22]. Although the dominant cost varies across queries (e.g., materialization in CQ1 and exploration in CQ3 and CQ4), a substantial fraction of GPU and CPU work is overlapped. These results demonstrate that BIM effectively reduces overall execution time by overlapping heterogeneous computation.

Table 6. GPU utilization metrics (SF=10).

Query	Active warps per scheduler	Active threads per warp	DRAM throughput	L2 hit rates
Q1	46.1 %	75.0 %	0.3 %	93.3 %
Q2	45.8 %	72.1 %	0.7 %	90.2 %
Q3	45.9 %	72.3 %	0.8 %	89.8 %
Q6	38.2 %	67.5 %	0.6 %	89.7 %

Table 7. Query-level parallelism characteristics.

Dataset	Query	# TGs	Max. TG depth	Max. hops	Fan-out in base-TGs
SF=10	Q1	37	3	15	18
	Q5	1,423	2	10	1,388
Span=1Y	Q1	28	4	20	9
	Q5	16	4	20	5
	Q8	255	6	30	66

Table 8. Breakdown of BIM execution time (SF=10).

Query	GPU (sec)	CPU (sec)	BIM total (sec)	Overlap ratio
CQ1	46.9	205.75	206.34	98.8 %
CQ3	7.78	2.75	8.75	64.8 %
CQ4	18.07	3.65	19.18	69.6 %

8.5 Ablation Study on cuRPQ

Figure 17(a) shows the query execution times with **varying segment buffer sizes**. As shown earlier in Figure 13(b), Q5 at SF=10 requires up to 57.2 GB of memory. When the segment buffer size decreases, sub-TG partitioning occurs more frequently and the number of bridge segments to be managed increases, introducing additional overhead and gradually increasing execution time. In particular, when the buffer size is reduced to as small as 10 GB, segments cannot be released until all sub-TGs for a starting vertex have been processed, which can lead to insufficient segments in the pool. In such cases, cuRPQ adopts the approach of temporarily reducing the batch size, which prevents full utilization of GPU parallelism and leads to performance degradation.

Figure 17(b) shows the query execution times with **varying UR buffer sizes**. Since cuRPQ builds slices asynchronously through BIM, it shows only a negligible difference in execution time

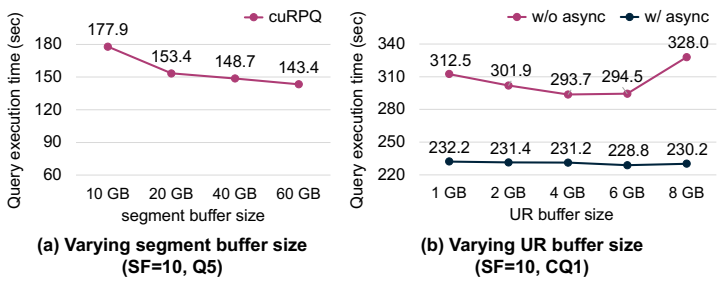


Fig. 17. Impact of varying GPU buffer sizes.

across UR buffer sizes. If slice building were performed synchronously, the execution time would increase by up to 1.4X at 8 GB compared to the asynchronous case. In general, larger UR buffers improve performance by reducing the number of kernel suspensions and resumptions. However, excessively large buffers transfer more results to the CPU at once, leading to the activation of more temporary slice buffers simultaneously. This causes scattered writes and lowers cache efficiency, which in turn reduces performance.

9 Related Work

- **Optimization techniques for RPQ evaluation:** In automata-based approaches, RPQ evaluation has been optimized by decomposing regular expressions or selecting cost-effective sub-RPQs [7, 45, 60]. For multi-query optimization, techniques have been proposed to reduce computational cost by sharing common parts across multiple RPQs [2, 3, 57, 64]. Since cuRPQ provides a flexible plan representation, it can naturally incorporate these optimizations to achieve further performance improvements.
- **Optimization techniques for transitive closure:** Several approaches have accelerated transitive closure computation based on Warshall's algorithm [81] using Boolean algebra [8, 42, 67], and GPU algorithms have been explored to address bottlenecks from irregular memory access and thread synchronization [37, 71]. cuRPQ can flexibly integrate these techniques with the loop-cache plan to further improve transitive closure performance.
- **Persistent RPQ in streaming graph processing:** Persistent RPQs have been studied in streaming graphs, which monitor pre-registered queries and update results incrementally [35, 62, 63], commonly using datasets such as LDBC SNB and StackOverflow. For LDBC SNB, evaluations adopt fine-grained windows of 3 days with sliding intervals of 6 hours [35]. Although cuRPQ does not directly support persistent RPQs, it efficiently processes large-scale snapshots (e.g., entire graphs or yearly intervals) and demonstrates strong performance even on challenging datasets.

10 Conclusion and Discussion

In this paper, we proposed cuRPQ, a GPU-optimized framework that preserves the automata-based exploration paradigm while efficiently supporting CRPQs, which are traditionally handled inefficiently by algebra-based methods. To avoid the substantial data migration and synchronization overhead incurred by transparent GPU memory spilling during materialization, cuRPQ explicitly decouples GPU-based exploration from CPU-side materialization, enabling stable and high-throughput execution. Extensive experiments demonstrated that cuRPQ successfully handled queries producing trillions of results, achieving up to 4,945X and 269X speedups over state-of-the-art algebra-based and automata-based methods, respectively. Beyond its current design, cuRPQ can theoretically be extended to support RPQs with additional constraints such as length bounds and word-equality or prefix/suffix relations. Length constraints can be naturally enforced by controlling traversal depth, while word-equality or prefix/suffix relations would require comparing label sequences across multiple paths during traversal.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00347471) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW).

References

[1] Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2025. Output-Sensitive Evaluation of Regular Path Queries. *Proceedings of the ACM on Management of Data* 3, 2 (2025), 1–20.

[2] Zahid Abul-Basher. 2017. Multiple-query optimization of regular path queries. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1426–1430.

[3] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, and Mark H Chignell. 2016. SwarmGuide: Towards Multiple-Query Optimization in Graph Databases.. In *AMW*.

[4] Rakesh Agrawal. 2002. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering* 14, 7 (2002), 879–885.

[5] Rakesh Agrawal and HV Jagadish. 1987. Direct Algorithms for Computing the Transitive Closure of Database Relations.. In *VLDB*, Vol. 87. 1–4.

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[7] Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2024. Optimizing RPQs over a compact graph representation. *The VLDB Journal* 33, 2 (2024), 349–374.

[8] Diego Arroyuelo, Adrián Gómez-Brandón, and Gonzalo Navarro. 2025. Evaluating regular path queries on compressed adjacency matrices. *The VLDB Journal* 34, 1 (2025), 2.

[9] 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.

[10] Pablo Barceló and Pablo Muñoz. 2017. Graph logics with rational relations: the role of word combinatorics. *ACM Transactions on Computational Logic (TOCL)* 18, 2 (2017), 1–41.

[11] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 175–188.

[12] Georgiy Belyanin, Semyon Grigoriev, and Rodion Suvorov. 2024. Single-Source Regular Path Querying in Terms of Linear Algebra. *arXiv preprint arXiv:2412.10287* (2024).

[13] Mauro Bisson and Massimiliano Fatica. 2017. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510.

[14] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2022. *Querying graphs*. Springer Nature.

[15] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the maze of Wikidata query logs. In *The World Wide Web Conference*. 127–138.

[16] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.

[17] Katrin Casel and Markus L Schmid. 2023. Fine-grained complexity of regular path queries. *Logical Methods in Computer Science* 19 (2023).

[18] NVIDIA Corporation. 2025. RAPIDS. https://rapids.ai.

[19] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. *ACM SIGMOD Record* 16, 3 (1987), 323–330.

[20] Tamara Cucumides, Juan Reutter, and Domagoj Vrgoč. 2023. Size bounds and algorithms for conjunctive regular path queries. In *26th International Conference on Database Theory (ICDT 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13–1.

[21] Alan Davoust and Babak Esfandiari. 2016. Processing regular path queries on arbitrarily distributed data. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 844–861.

[22] Alexandre Denis and François Trahay. 2016. Mpi overlap: Benchmark and analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 258–267.

[23] Alin Deutsch, Nadime Francis, Keith Green, Alastair anzinn2016generald Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

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

[25] Jens Dörpinghaus, Sonja Klante, Martin Christian, Christof Meigen, and Carsten Düing. 2022. From social networks to knowledge graphs: A plea for interdisciplinary approaches. *Social Sciences & Humanities Open* 6, 1 (2022), 100337.

[26] Jens Dörpinghaus, Andreas Stefan, Bruce Schultz, and Marc Jacobs. 2022. Context mining and graph queries on giant biomedical knowledge graphs. *Knowledge and Information Systems* 64, 5 (2022), 1239–1262.

[27] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8.

[28] Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Jinsoo Lee, Jakub Yaghob, Sungpack Hong, and Hassan Chafi. 2023. Distributed Asynchronous Regular Path Queries (RPQs) on Graphs. In *Proceedings of the 24th International Middleware Conference: Industrial Track*. 35–41.

[29] 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.
[30] Diego Figueira, Adwait Godbole, S Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of simple regular path queries. *arXiv preprint arXiv:2003.04411* (2020).
[31] Diego Figueira, Rémi Morvan, and Miguel Romero. 2025. Minimizing Conjunctive Regular Path Queries. *Proceedings of the ACM on Management of Data* 3, 2 (2025), 1–25.
[32] Diego Figueira and Miguel Romero. 2023. Conjunctive Regular Path Queries under Injective Semantics. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 231–240.
[33] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. Database Theory in Action: Cypher, GQL, and Regular Path Queries. In *28th International Conference on Database Theory (ICDT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 36–1.
[34] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.
[35] Xiangyang Gou, Xinyi Ye, Lei Zou, and Jeffrey Xu Yu. 2024. LM-SRPQ: Efficiently Answering Regular Path Query in Streaming Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1047–1059.
[36] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying.. In *EDBT*. 520–523.
[37] Oded Green, Zhihui Du, Sanyamee Patel, Zehui Xie, Hang Liu, and David A Bader. 2021. Anti-section transitive closure. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 192–201.
[38] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.
[39] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 511–524.
[40] HeavyDB. 2025. HeavyDB. https://github.com/heavyai/heavydb.
[41] Nikolaos Karalis, Alexander Bigerl, Liss Heidrich, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. 2024. Efficient evaluation of conjunctive regular path queries using multi-way joins. In *European Semantic Web Conference*. Springer, 218–235.
[42] Gary J Katz and Joseph T Kider. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (GH'08)*. 47–55.
[43] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++ A scalable and fast graph analytics system. In *Proceedings of the 2018 international conference on management of data*. 395–410.
[44] Ioannis Konstas, Vassilios Stathopoulos, and Joemon M Jose. 2009. On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. 195–202.
[45] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *International Conference on Scientific and Statistical Database Management*. Springer, 177–194.
[46] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* (2022), 1–25.
[47] Ulf Leser. 2005. A query language for biological networks. *Bioinformatics* 21, suppl_2 (2005), ii33–ii39.
[48] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[49] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2016. Network motif discovery: A GPU approach. *IEEE transactions on knowledge and data engineering* 29, 3 (2016), 513–528.
[50] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. {NeuGraph}: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.
[51] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.
[52] Wim Martens and Tina Trautner. 2019. Dichotomies for evaluating simple regular path queries. *ACM Transactions on Database Systems (TODS)* 44, 4 (2019), 1–46.
[53] Alberto O Mendelzon and Peter T Wood. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.
[54] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–11.

[55] Kento Miura, Toshiyuki Amagasa, Hiroyuki Kitagawa, R Bordawekar, and T Lahiri. 2019. Accelerating Regular Path Queries using FPGA.. In *ADMS@ VLDB*. 47–54.
[56] Thomas Mulder, George Fletcher, and Nikolay Yakovets. 2025. Optimizing navigational graph queries. *The VLDB Journal* 34, 2 (2025), 16.
[57] Inju Na, Yang-Sae Moon, Ilyeop Yi, Kyu-Young Whang, and Soon J Hyun. 2022. Regular path query evaluation sharing a reduced transitive closure based on graph reduction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1675–1686.
[58] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.
[59] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. (2012).
[60] Van-Quyet Nguyen and Kyungbaek Kim. 2017. Efficient regular path query evaluation by splitting with unit-subquery cost matrix. *IEICE TRANSACTIONS on Information and Systems* 100, 10 (2017), 2648–2652.
[61] Seyeon Oh, Heeyong Yoon, Donghyoung Han, and Min-soo Kim. 2025. GFlux: A Fast GPU-Based Out-of-Memory Multi-Hop Query Processing Framework for Trillion-Edge Graphs. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 932–945.
[62] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.
[63] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2022. Evaluating complex queries on streaming graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 272–285.
[64] Yue Pang, Lei Zou, Jeffrey Xu Yu, and Linglin Yang. 2024. Materialized View Selection & View-Based Query Planning for Regular Path Queries. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[65] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2024. INFINEL: An efficient GPU-based processing method for unpredictable large output graph queries. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 147–159.
[66] 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.
[67] Gerald Penn. 2006. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science* 354, 1 (2006), 72–81.
[68] Cynthia Phillips and Laura Painton Swiler. 1998. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*. 71–79.
[69] Alexandra Poulovassilis, Petra Selmer, and Peter T Wood. 2016. Approximation and relaxation of semantic web path queries. *Journal of Web Semantics* 40 (2016), 1–21.
[70] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.
[71] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2023. Towards iterative relational algebra on the {GPU}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 1009–1016.
[72] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.
[73] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC social network benchmark: Business intelligence workload. *Proceedings of the VLDB Endowment* 16, 4 (2022), 877–890.
[74] Bryan Thompson, Mike Personick, and Martyn Cutcher. 2016. The Bigdata® RDF graph database. In *Linked data management*. Chapman and Hall/CRC, 221–266.
[75] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).
[76] Jacques Van Helden, Avi Naim, Renato Mancuso, Mattew Eldridge, Lorenz Wernisch, David Gilbert, and Shoshana J Wodak. 2000. Representing and analysing molecular and cellular function in the computer. *Biological chemistry* 381, 9-10 (2000), 921–935.
[77] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.
[78] W3C. 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query.
[79] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the 2019 international conference on management of data*. 1463–1480.
[80] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management*

of Data. 1301–1315.
[81] Stephen Warshall. 1962. A theorem on boolean matrices. *Journal of the ACM (JACM)* 9, 1 (1962), 11–12.
[82] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.
[83] MV Wilkes. 1974. The art of computer programming, volume 3, sorting and searching.
[84] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[85] Peichen Xie, Zhigao Zheng, Yongluan Zhou, Yang Xiu, Hao Liu, Zhixiang Yang, Yu Zhang, and Bo Du. 2025. GPU Architectures in Graph Analytics: A Comparative Experimental Study. (2025).
[86] Zequan Xu, Siqiang Luo, Jieming Shi, Hui Li, Chen Lin, Qihang Sun, and Shaofeng Hu. 2022. Efficiently answering k-hop reachability queries in large dynamic graphs for fraud feature extraction. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 238–245.
[87] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data*. 1875–1889.
[88] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster depth-first subgraph matching on gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3151–3163.
[89] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1249–1260.
[90] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.
[91] Daniel Zinn, Haicheng Wu, Jin Wang, Molham Aref, and Sudhakar Yalamanchili. 2016. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. 12–21.

Received October 2025; revised January 2026; accepted February 2026