RESEARCH-ARTICLE

DRPQ: Distributed Evaluation of Regular Path Queries On Streaming Graphs

SIYUAN ZHANG, Fudan University, Shanghai, China

KAI ZHANG, Fudan University, Shanghai, China

ZHENGYING HE, Fudan University, Shanghai, China

YINAN JING, Fudan University, Shanghai, China

ZHIGANG ZHAO, Fudan University, Shanghai, China

XIAOYANGSEAN WANG, Fudan University, Shanghai, China

Proceedings of the ACM on Management of Data, Volume 3, Issue 6 (December 2025)
https://doi.org/10.1145/3769782
EISSN: 2836-6573

DRPQ: Distributed Evaluation of Regular Path Queries On Streaming Graphs

SIYUAN ZHANG, Fudan University, China
KAI ZHANG, Fudan University, China
ZHENYING HE, Fudan University, China
YINAN JING, Fudan University, China
ZHIGANG ZHAO, Fudan University, China
X. SEAN WANG, Fudan University, China

Persistent Regular Path Query (RPQ) on streaming graphs is widely applicable to many online analysis applications. Existing research primarily focuses on the single-worker scenario, while scaling out to distributed RPQ processing on multiple workers is desirable when facing a high workload. Existing distributed solutions are designed for general streaming queries, and various bottlenecks exist that significantly limit the performance when performing streaming RPQ evaluation. The challenge is how to execute queries with multiple workers while introducing limited overhead and ensuring sufficient speedup as the number of workers increases.

This paper introduces a distributed processing strategy called DRPQ by carefully dividing a query into multiple partially matched query tasks. The idea is to form query tasks based on initial matches of the graph against the given regular expression, and to dynamically distribute these tasks to workers to balance their workloads. To reduce redundant evaluation across different workers, a grouping method is proposed to find query tasks that are likely to share evaluation processes, and send them to the same workers. Extensive experiments on two real-world graph datasets demonstrate that DRPQ is significantly more efficient and scalable than existing distributed solutions. Furthermore, the proposed grouping method proves to be particularly effective, nearly doubling the throughput in most cases.

CCS Concepts: • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: Distributed Evaluation, Regular Path Query, Streaming Graph

ACM Reference Format:
Siyuan Zhang, Kai Zhang, Zhenying He, Yinan Jing, Zhigang Zhao, and X. Sean Wang. 2025. DRPQ: Distributed Evaluation of Regular Path Queries On Streaming Graphs. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 317 (December 2025), 27 pages. https://doi.org/10.1145/3769782

1 Introduction

A persistent Regular Path Query (RPQ) on a streaming graph is to continuously find every pair of vertices that are connected by a path, such that the edge label sequence of this path matches a given regular expression. It is typically addressed in the sliding window model so that the edges in the matched path are required to be in a single window, and consecutive windows often overlap a lot.

Authors' Contact Information: Siyuan Zhang, siyuanzhang17@fudan.edu.cn, Fudan University, Shanghai, China; Kai Zhang, zhangk@fudan.edu.cn, Fudan University, Shanghai, China; Zhenying He, zhenying@fudan.edu.cn, Fudan University, Shanghai, China; Yinan Jing, jingyn@fudan.edu.cn, Fudan University, Shanghai, China; Zhigang Zhao, zgzhao21@m.fudan.edu.cn, Fudan University, Shanghai, China; X. Sean Wang, xywangCS@fudan.edu.cn, Fudan University, Shanghai, China.

ACM 2836-6573/2025/12-ART317
https://doi.org/10.1145/3769782

RPQ on streaming graphs is widely applicable in various online analysis applications, including social network analysis [16, 27, 40] and network security monitoring [35]. Figure 1(a) illustrates an example of credit-card-fraud activities detection. Accounts are represented as vertices, while transactions between accounts are edges. Each transaction is labeled with a type and a timestamp. The dotted lines represent a kind of the credit-card-fraud model, which can be defined as query $R = (Credit\ pay \cdot transfer^{+})$ with the restriction that all edges fall within the same sliding window W $(max(t_1, \cdots, t_n) - min(t_1, \cdots, t_n) \leq |W|)$. If both ends of a matched path are the same account, then there is probably a fraud.

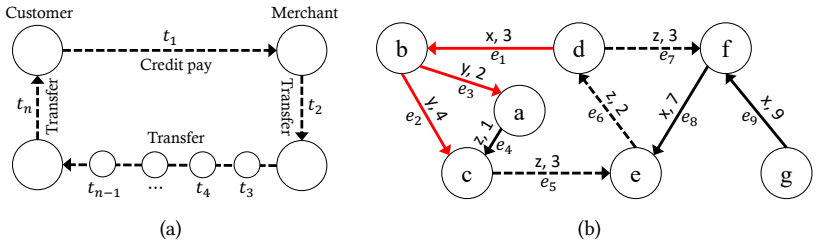


Fig. 1. (a) a graph of a credit-card-fraud example where the nodes are accounts and the edges are transactions, (b) a graph representation of a streaming graph.

RPQs on static graphs have been extensively studied [18, 32, 34, 41]. However, streaming graphs bring significant new challenges. The difference is that in the streaming scenario, there is usually a large overlap of edges between consecutive windows. To efficiently process RPQs on streaming graphs, the algorithm in Pacaci et al. [45] continuously maintains a set of spanning trees to gradually construct query results as the edges arrive. The idea is to keep in the spanning trees all the initial partial and full matches of the given regular expression found so far in the streaming graph, each starting with a root node that is the beginning of the matches. This strategy avoids the full graph traversal with each graph update. Each time the window slides forward, an expiration process occurs that removes all the edges that are out of the window, along with all the edges on the affected paths. Zhang et al. [59] identified that the expiration process needs a forced blocking phase which greatly degrades the query performance. They proposed a new data structure called Timestamped Rooted Digraph (TRD for short) which adopts a multi-window parallel processing strategy to remove the blocking phases and provides more opportunities for parallel processing, achieving much better performance.

The aforementioned algorithms and their extensions [25] are all for the single-worker scenario. When the workload exceeds the worker's processing capability (probably caused by the arrival of data that is large volumes with high velocity), an efficient and scalable distributed solution is desirable but currently missing. Existing distributed solutions for streaming graph queries mostly fall into three categories. (1) Split the full query into multiple subqueries and evaluate different subqueries in different workers [46]. With this solution, join or merge operations are needed for producing final results. (2) Divide the streaming graph into multiple distinct subgraphs based on some streaming graph partitioning methods [24]. With this strategy, communication among workers is needed when processing the cross-worker edges/vertices. (3) Divide the streaming graph into multiple distinct time-based or count-based windows (e.g., Apache Flink, Spark Streaming, and Timely Dataflow). With this solution, cross-node communication is avoided and data in overlapping windows are processed independently in parallel.

The above distributed solutions are general-purpose aiming at general streaming graph query applications (e.g., Subgraph Matching, Triangle Counting) and their generality comes at a performance cost for evaluating streaming RPQs. Strategy (1) requires computing results for all sub-regular expressions even if most of them do not participate in any final results, and its scalability is inherently limited by how many parts the regular expression can be split into. Strategy (2) has a significant communication overhead among workers since the path traversal process may reach a large part of the graph especially when dealing with transitivity-like queries (using "*" or "+") or dense graphs (e.g., the Stack Overflow dataset). Additionally, maintaining load balance through graph repartitioning will introduce blocking phases that severely degrade system performance. Strategy (3) leads to redundant query processing due to the large overlap of edges between consecutive windows. Effectively distributing the RPQ workload with minimal overhead while achieving a scalable speedup as the number of workers increases remains a challenge, and our solution directly addresses this issue.

In this paper, we propose DRPQ, a distributed processing strategy for RPQs that deviates from all three strategies mentioned above, which partitions query instantiations into multiple "partially instantiated queries" that can be executed in parallel across workers. Each partially instantiated query represents a RPQ evaluation task, beginning from a predetermined path (termed a "partial match") whose path label matches the beginning of the given regular expression. The goal of the task to extend the partial match to discover all valid paths that satisfy the query condition. We only need to send all the edges in the sliding windows that the partial match belongs to, along with the initial partial matches. These form a "partially matched query task" (PMQT for short). Each PMQT serves as a "unit of work" and can then be processed independently by any worker, following the same RPQ evaluation approach as in [45] or [59]. The length of the selected partial match affects the granularity of parallelism: longer partial matches result in a greater number of smaller PMQTs, facilitating better scalability across workers but increasing the overhead of partial match selection. An example is shown in Figure 1(b). Consider the regular expression $R = (x \circ y \circ z^*)$ and a sliding window size $|W| = 3$. The path $e_1 \cdot e_2$, labeled $x \circ y$, serves as a partial match. Its associated PMQT consists of edges e_1, e_2, e_3, e_5, e_6 and e_7, with timestamps falling within range $[2, 5]$ (computed as $(ts(e_1) \pm |W|) \cap (ts(e_2) \pm |W|)$ where $ts(e)$ is the timestamp of e). Intuitively, if a matching path includes $e_1 \cdot e_2$, then all its edges must be among those within this PMQT. Hence, each PMQT can be used as a unit of work to find all the results of the RPQ query.

DRPQ properly addresses the challenges of the existing distributed implementation strategies and develop an efficient distributed extension of existing streaming RPQ evaluation methods. Firstly, it eliminates communication overhead among workers (compared to strategy (1) & (2)). Secondly, the same method in [45] or [59] can be adopted in DRPQ when evaluating PMQTs to reduce the extraneous processing that is not involved in the final query results (compared to (1)) and avoid the redundant computation caused by overlapping sliding windows (compared to (3)). Thirdly, DRPQ scales efficiently by distributing PMQTs across multiple workers (compared to (1)), and achieve load balancing through careful PMQT allocation without introducing blocking phases (compared to (2)).

Although DRPQ provides an efficient solution for distributed evaluating streaming RPQ, this strategy suffers from the fact that the edges may be duplicate for different PMQTs in different workers, and PMQTs assigned to different workers may lose the opportunity to share some traversal process of the same edges when common subgraphs exist. For example, as shown in Figure 1(b), both PMQTs with partial match e_1, e_2 and e_1, e_3 contain the same sub-expression z^* ($x \circ y$ is already matched by their own partial matches). The edges e_5, e_6, and e_7 can form a path whose label sequence matches z^* (marked with the dotted line in the figure) and can be shared by these two

PMQTs. When such PMQTs are assigned to different workers, the above traversal process will be duplicated. Reducing this redundant computation can increase the overall efficiency.

In this paper, we propose a novel grouping strategy to recognize the features of the PMQTs that may share traversal process and then use a quick heuristic method to put possibly "sharing" PMQTs into groups. The grouping strategy needs to be tailored towards the particular RPQ evaluation method. In this paper, we use the method in [59] for this purpose. The PMQTs in the same group are then prioritized to send to the same workers to reduce the aforementioned redundant computation and potentially reduce the overlap of streaming graph edges that are sent to different workers.

We carry out extensive experiments with two representative real-world datasets to evaluate the efficiency and scalability of all the aforementioned distributed strategies. The result shows that DRPQ has a much higher throughput compared to the other strategies, and that adopting grouping strategy in DRPQ can greatly improve the corresponding throughput (almost doubled in many cases). We also evaluate the impact of the selected features of the PMQTs.

To the best of our knowledge, DRPQ is the first efficient and scalable distributed processing strategy for streaming RPQs in the literature. Here we highlight our contributions as follows:

- We propose a distributed processing strategy for scalable and efficient streaming RPQ evaluation by decomposing query instantiations into multiple PMQTs, each processed as an independent unit of work.
- We introduce a grouping strategy to reduce redundant computations by increasing the likelihood that PMQTs sharing common traversal process are assigned to the same worker.
- We conduct extensive experiments to evaluate our distributed strategy DRPQ for its efficiency and scalability.

The remainder of the paper is structured as follows. Section 2 presents the problem definition and a brief description of algorithms leveraging TRDs. Section 3 provides a description of the task unit PMQT and the DRPQ strategy. Section 4 introduces our grouping strategy and gives a detailed discussion about the features of the PMQTs that may share traversal process. Section 5 shows the experimental results with a detailed analysis. Section 6 describes the related works. Finally, Section 7 summarizes the paper.

2 Preliminaries

In this section, we first provide a formal definition of the streaming graph and streaming RPQ task. Then, we describe how to maintain TRDs for streaming RPQ evaluation in the single-worker scenario. Most of the following definitions are adopted from [45] and [59].

2.1 Streaming RPQ Evaluation

Definition 1. *Streaming Graph: A* streaming graph *refers to a sequence of time-ordered tuples* $S = \tau_1, \tau_2, \cdots, \tau_n$, *where each tuple* $\tau_i = (e_i, t_i)$ *represents a directed edge* $e_i = (v_x, v_y, l_i)$ *from source vertex* v_x *to target vertex* v_y *with edge label* $\phi(e_i) = l_i$ *and arrival timestamp* $ts(\tau_i) = t_i$.

Figure 2(a) shows a streaming graph example consisting of 8 tuples with 3 kinds of edge labels (x, y and z) and 7 vertices (a to g). We follow the same assumption as in [45, 59] that all tuples arrive in the order of their timestamps, i.e., the timestamps of tuples τ_i and τ_j in S satisfy $ts(\tau_i) \leq ts(\tau_j)$ for all $i < j$. We also assume that the minimum timestamp of the tuples in the stream is 0.

Definition 2. *Path and Path Label: A path that connects vertices* v_x *and* v_y *is a sequence of directed edges* $p = e_1, e_2, \cdots, e_n$ *where* v_x *is the source vertex of* e_1 *and* v_y *is the target vertex of* e_n, *and the target vertex of* e_i *is the same as the source vertex of* e_{i+1} *for each* $i \in [1, n-1]$. *The corresponding path label* $\phi(p)$ *of path* p *is the label sequence of the edges in* p $\phi(p) = \phi(e_1), \phi(e_2), \cdots, \phi(e_n)$.

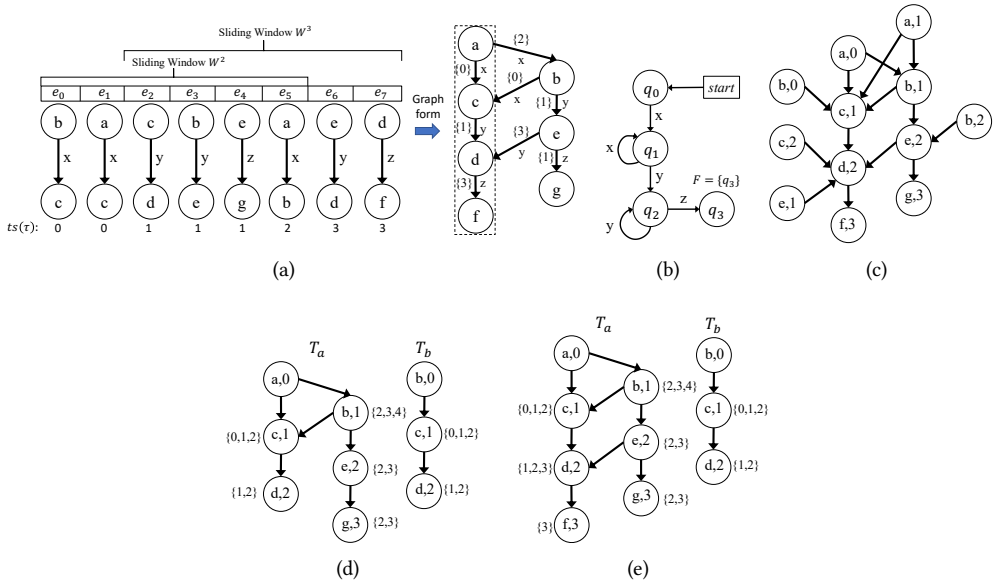


Fig. 2. (a) a streaming graph S (the left part) consisting of two sliding windows W^2 and W^3 (marked in the graph) with window size $|W| = 3$. The right part in (a) is its graph representation where the timestamps $ts(\tau)$ and labels are all marked next to the edges, (b) a DFA for query expression $R = (x^+ \circ y^+ \circ z)$. (c) is the product graph of S at time $t = 3$, where the mapped tuples of the edges are omitted. (d) & (e) are the generated TRDs in S at time 2 and 3 respectively.

Figure 2(a) shows the streaming graph in the tuple sequence form and in the graph form, respectively. An example path p in the streaming graph is boxed with dotted lines in Figure 2(a), which consists of 3 edges e_1, e_2, e_7 with path label $\phi(p) = x, y, z$.

In applications, we usually adopt a time-based sliding window (sliding window for short) to restrict the tuples in a streaming graph to be used in forming a particular query results.

Definition 3. *Sliding Window over Streaming Graph: For a given streaming graph S, a sliding window W at time t, denoted W^t, with window size $|W|$ and sliding step length $Step(W)$, represents the set of tuples τ_i, where $\tau_i \in S$ and $ts(\tau_i) \in (t-|W|, t]$. In the above, t must be equal to $|W|+k*Step(W)-1$ for some integer $k \geq 0$.*

The edges of the tuples in a sliding window form a "snapshot graph" and the snapshot graph at time 2 with sliding window $|W| = 3$ and $Step(W) = 1$ consists of edges e_0 to e_5 in graph 2(a). The first sliding window is $W^{|W|-1}$ ($|W| \geq 1$). Note that we usually assume $Step|W| \leq |W|$.

Based on the definition of the sliding window, we now describe the streaming RPQ evaluation task.

Definition 4. *Streaming RPQ: Let Σ be a finite alphabet set consisting of all the labels of edges in a streaming graph S, and $L(R)$ ($L(R) \neq \emptyset$) be the set of label sequences that match a given regular expression R over Σ. A streaming RPQ task on a streaming graph S with regular expression R and sliding window W (with $|W|$ and $Step(W)$ understood) is to find all the vertex pairs (u, v) such that there exists a non-empty path p from vertex u to v satisfying $\phi(p) \in L(R)$ in the snapshot graph for the sliding window W^t for an integer $t = |W| + k * Step(W) - 1$, where $k \geq 0$ is an integer.*

For a given sliding window W^t, the above streaming RPQ can be formally defined as $Q^t_{R,W}$. Then,

$$Q^t_{R,W} = \{(u,v) | \text{there exists a path } p \text{ in } S \text{ from vertex } u \text{ to } v \text{ such that } \phi(p) \in L(R) \text{ and } t - |W| < ts(\tau_i) \leq t \text{ for each tuple } \tau_i \text{ in } p\}$$

$Q^t_{R,W}$ may be simplified as Q^t if both the regular expression R and sliding window W are understood.

Here we provide an example to illustrate the evaluation task.

Example 1. *Consider the RPQ evaluation with regular expression $R = (x^+ \circ y^+ \circ z)$ and sliding window W with $|W| = 3$ and $Step(W) = 1$ on the streaming graph S in Figure 2(a). The corresponding snapshot graph in W^2 consists of edges e_0 to e_5 in graph 2(a). There is only one path $p = e_5, e_3, e_4$ whose label sequence $\phi(p) = x, y, z$ matches R. Thus the streaming RPQ evaluation result in $Q^2 = \{(a, g)\}$.*

When the sliding window moves to W^3, the edge e_1 is out of the current sliding window, and $Q^3 = \{(a, g), (a, f)\}$ holds by the two paths e_5, e_3, e_4 and e_5, e_3, e_6, e_7, respectively.

2.2 The Multi-Window Parallel Strategy

We now introduce multi-window parallel method that executes streaming RPQ leveraging TRDs [59]. We begin by a formal definition of the *Deterministic Finite Automaton* for regular expressions.

Definition 5. *Deterministic Finite Automaton (DFA): Given a regular expression R, a deterministic finite automaton (DFA) for R refers to a 5-tuple $DFA(R) = (K, L, \delta, q_0, F)$, where (i) K is a finite set of states, (ii) L is a finite set of symbols, (iii) mapping function $\delta: K \times L \rightarrow K$, (iv) q_0 is the initial state of DFA, and F is a set of final states such that $q_0 \in K$ and $F \subseteq K$. It is required that $DFA(R)$ accepts exactly all the sequences in $L(R)$.*

For a given regular expression R, we first construct a nondeterministic finite automaton (NFA) using Thompson's construction algorithm [54], and then convert it into a deterministic finite automaton (DFA) using Hopcroft's algorithm [28]. This transformation is consistent with prior works and makes the automaton more predictable in real-time performance under tight resource constraints. An example DFA for $R = (x^+ \circ y^+ \circ z)$ is shown in Figure 2(b). Although the size of the DFA can be large in theory, it is has not been a practical bottleneck in our current experimental settings.

Based on the deterministic finite automaton (DFA), we introduce the product graph [21, 38], which transforms the regular expression matching process on a graph into a path traversal problem over a constructed graph. The product graph construction has been widely adopted as a fundamental mechanism for evaluating regular path queries (RPQs) in prior works.

Definition 6. *Product Graph: Given a $DFA(R) = (Q, \Sigma, \delta, q_0, F)$ and a snapshot graph $G = (V, E, \phi)$ in streaming graph S with V being the vertex set and each edge $e \in E$ labeled by $\phi(e) \in \Sigma$. The* product graph *is defined as $P_{G,DFA(R)} = (V_\times, E_\times)$ where $V_\times = V \times Q$, $E_\times \subseteq V_\times \times V_\times$, and $((u, s_u), (v, s_v))$ is in $E_\times$ iff edge $(u, v, l) \in E$ and $\delta(s_u, l) = s_v$. Each edge $((u, s_u), (v, s_v)) \in E_\times$ maps to the set of tuples $((u, v, l), t) \in S$ where $\phi(s_u, l) = s_v$.*

Figure 2(c) illustrates the product graph constructed from the streaming graph S in Figure 2(a), guided by the $DFA(R)$ shown in Figure 2(b). Each edge in the product graph corresponds to a set of tuples. For example, the product edge $((a, 0), (c, 1))$ maps to the tuple set $\{(e_1, 0)\}$. We leverage the product graph to streamline the technical exposition and to enable incremental construction of the Timestamped Rooted Digraph (TRD) as new edges arrive in the stream. A TRD can be viewed as a timestamp-annotated, rooted subgraph of the product graph, specifically tailored for streaming RPQ evaluation under sliding window semantics.

Definition 7. *Timestamped Rooted Digraph (TRD): For a given regular expression R and a streaming graph S, a timestamped rooted digraph for the constructed product graph $P_{G,DFA(R)}$ is a connected directed graph consisting of nodes in $P_{G,DFA(R)}$. Each TRD has a root node (x, q_0), and such TRD is denoted as T_{x,q_0} (T_x for short since q_0 is determined). Each edge in T_x is selected from $P_{G,DFA(R)}$. No two nodes in a TRD have the same vertex and state. Each node (u, s_u) in T_x is associated with a non-empty finite set of non-negative integers denoted $(u, s_u).ts$, which is called its timestamps. The timestamp of the root contains all the non-negative integers by default. Each ts associated with a node has the property that for each $t \in ts$ there exists a path from the root to itself such that t is one of the timestamps of each node on the path.*

A TRD is required to be rooted, and the root must have at least one directed path to every other node it contains. Intuitively, a TRD incrementally maintains all paths in the product graph $P_{G,DFA(R)}$ that correspond to partial or complete matches of R over the streaming graph S within a given sliding window. Specifically, if a timestamp t appears in the timestamp set of a non-root node, then for each edge along the path from the root to that node, there must exist at least one matching stream tuple (e_i, t_i) corresponding to the product edge such that $t - |W| < t_i \leq t$. Therefore, a pair (u, v) appears in the query result Q^t if and only if (u, q_0) is the root and (v, s_v) is a node in the TRD with $s_v \in F$ and $t \in (v, s_v).ts$.

An example of a TRD T_a is shown in Figure 2(d), where the labeled integer set next to each node indicates its associated timestamps. Here, we denote states q_0 to q_3 in $DFA(R)$ as 0 to 3, respectively. Node $(b, 1)$ is labeled with timestamps $\{2, 3, 4\}$ because it is reached via edge e_5, which arrives at $t = 2$ and remains valid in the sliding windows W^2, W^3, and W^4 under a window size of $|W| = 3$. In general, a tuple with timestamp t is retained in all windows $W^{t'}$ where $t' \in [t, t + |W| - 1]$. When a node such as $(g, 3)$ in this figure has a state $3 \in F$, a result pair (a, g) is produced. That is, the edge label sequence x, y, z from $(a, 0)$ to $(g, 3)$ belongs to $L(R)$, and the timestamps $\{2, 3\}$ of node $(g, 3)$ indicate the sliding windows W^2 and W^3 in which the result pair holds.

The product graph is incrementally updated as tuples arrive in the stream, and the method in [59] continuously creates and updates the TRDs to reflect changes in the product graph and capture all initial partial matches of R under sliding window semantics. It employs the following three rules to maintain TRDs when a tuple $((u, v, l), ts)$ arrives (assuming the product graph has already been updated):

(1) Create a new TRD T_u if it does not already exist and if (u, q_0) exists in $P_{G,DFA(R)}$.
(2) For each newly added edge $((u, s_u), (v, s_v))$ in $P_{G,DFA(R)}$, proceed as follows: if (v, s_v) is not already in any TRD, add it to a TRD T_x along with the edge $((u, s_u), (v, s_v))$ if $(u, s_u) \in T_x$ and $(u, s_u).ts \cap [ts, ts + |W|) \neq \emptyset$. Then, update the timestamps of the newly added or existing node (v, s_v) by setting $(v, s_v).ts = (v, s_v).ts \cup ((u, s_u).ts \cap [ts, ts + |W|))$, where $(v, s_v).ts$ is initialized as $\emptyset$ if (v, s_v) is newly added. This rule enables the TRD to grow via newly matched edges and propagates timestamps accordingly.
(3) If a new node is added or the timestamp set of any node is changed, reapply Rule (2) to all edges in $P_{G,DFA(R)}$. This rule ensures further extension of the TRD following any structural or timestamp changes.

Following the three rules, each node in a TRD records all root-to-it paths that partially or fully match R across multiple sliding windows. Note that multiple Step (2) can be performed in a TRD in parallel by adopting a careful lock, which effectively amortizes the expensive overhead. A timestamp t on a node indicates that at least one such path exists within window W^t. While the above assumes $Step(W) = 1$, it can be extended to $Step(W) > 1$ by considering the sliding steps when updating the timestamps in rule 2 accordingly.

Zhang et al. [59] optimize TRD maintenance and use a "dirty garbage collection" strategy to efficiently prune edges that won't contribute to future results. We refer readers to [59] for details, and discuss the garbage collection support of DRPQ in Section 3.4.2.

We provide an example to illustrate streaming RPQ using TRDs.

Example 2. *Consider the streaming RPQ evaluation in Example 1. The $DFA(R)$ of the regular expression $R = (x^+ \circ y^+ \circ z)$ is shown in Figure 2(b). Here we have $F = \{3\}$, and the states (q_0 to q_3) in $DFA(R)$ are 0 to 3. The window size $|W|$ is 3, and the sliding step length $Step(W)$ is 1. For the earliest sliding window W^2, edge e_0 and e_1 incurs a creation of TRD T_b and T_a (following rule 1), and insertion of the node $(c, 1)$ with timestamps $(c, 1).ts = \{0, 1, 2\}$ (rule 2). Then, e_2 is added into T_a and T_b (rule 2). The arrival of edges e_0, e_3, and e_4 does not cause any changes in TRD T_a until edge e_5 arrives and is inserted into T_a. In this case, following rule 3, edges e_0, e_3, and e_4 are recursively inserted into T_a, even though they have arrived earlier than e_5 did. The generated TRDs after e_5 arrives is shown in Figure 2(d). Note that there is a node $(g, 3)$ with timestamp $(g, 3).ts = \{2, 3\}$ in T_a whose state is in the final state set F, which means the query result (a, g) exists in both Q^2 and Q^3.*

Upon the arrival of edge e_6 and e_7, node $(d, 2)$ in T_a is reachable from root with two paths (e_1, e_2) and (e_5, e_3, e_6), and the timestamps of $(d, 2)$ is $(d, 2).ts = \{1, 2, 3\}$, which is calculated with $\{1, 2\} \cup (\{2, 3\} \cap \{3, 4, 5\})$ by rule 2. The final generated TRDs is shown in Figure 2(e). Note that node $(c, 1)$ in T_a is still retained even though edges e_0 and e_1 are out of the sliding window W^3 (garbage collection can be performed but omitted here in this paper). At this time, we have node $(f, 3)$ with timestamp $\{3\}$ and $(g, 3)$ with timestamps $\{2, 3\}$. We have (a, f) is in Q^3 and (a, g) is in Q^2 and Q^3.

3 Distributed Regular Path Query

In this section, we present the distributed implementation of streaming RPQs. We begin by defining *partial matches* and *partial matched query tasks*, then describe how to evaluate them individually and in groups. Finally, we introduce DRPQ, a system for distributed task evaluation designed to balance workload across workers.

3.1 Partially Matched Query Task (PMQT)

We begin by providing a formal definition of the *Partial Match*.

Definition 8. *Partial Match: For a given streaming graph S, sliding window $|W|$, and regular expression R. A partial match p is a path which consists of tuples $(e_1, t_1), (e_2, t_2), \cdots, (e_n, t_n)$ in S where $|t_i - t_j| \leq |W|$ holds for any $i, j \in [1, n]$ and edges $e_1, e_2, \cdots, e_n$ form a path whose path label partially matches R from the beginning. The sliding window set of p, denoted as $ts(p)$, contains timestamp t such that $\{t_1, \cdots, t_n\} \subseteq W^t$.*

An example partial match in Example 2 is $p = \{(e_1, 0), (e_2, 1)\}$ where edges e_1, e_2 form a path with label x, y that partially matches R from the start (i.e., the transition $q_0 \xrightarrow{x} q_1 \xrightarrow{y} q_2$ in Figure 2(b)). The sliding window set is $ts(p) = \{1, 2\}$. Based on it, we now define a *Partially Matched Query Task* for each partial match.

Definition 9. *Partially Matched Query Task (PMQT): For a given streaming graph S and a partial match $p = \{\tau_1, \tau_2, \cdots, \tau_n\}$, let t_{max} and t_{min} denote the maximum and minimum timestamps in $ts(p)$, a partially matched query task χ^p denotes the subset of all tuples τ' (including the tuples in p) in S whose timestamps satisfy:*

$$ts(\tau') \in [\lceil \frac{t_{max} - |W|}{Step(W)} \rceil * Step(W) + 1, \lfloor \frac{t_{min}}{Step(W)} \rfloor * Step(W) + |W| - 1]$$

Note if $(t_{max} - |W|)$ is negative, we take the left bound of $ts(\tau')$ as 0.

Here $W^{\lceil \frac{t_{max}-|W|}{Step(W)} \rceil * Step(W)+|W|}$ and $W^{\lfloor \frac{t_{min}}{Step(W)} \rfloor * Step(W)+|W|-1}$ are the first and last sliding windows that contain the partial match, respectively. An example of a PMQT with partial match $p = \{(e_5, 2)\}$ in the streaming graph in Figure 2(a) with sliding window $Step(W) = 1$ and $|W| = 2$ consists of the tuples $e_2, e_3, \cdots, e_7$ whose timestamps range from 1 to 3, and W^2 and W^3 are the first and the last sliding windows containing all tuples in the p, respectively.

For the regular expression R and a pre-given length threshold ℓen of the selected partial match, we continuously find partial matches consisting of ℓen tuples in a streaming graph and build PMQTs. That is, once partial matches are found, they are then assigned to workers along with the necessary subgraph for further processing.

Note that the partial matches starting from a vertex r can be identified by constructing TRD T_r, recording the length $\ell en'$ of each node from the root, and stopping recursive application of rule 2 when a node (u, s_u) reaches $\ell en' = \ell en$. Thus, the path from the root (r, q_0) to each such node (u, s_u) defines a specific partial match p. Here, we denote the TRD T_r as the associated TRD of χ^p, which by default consists of two nodes: root node (r, q_0) and initial node (u, s_u) (i.e., the start and end vertices of the partial match with states q_0 and s_u respectively) where $(u, s_u).ts = ts(p)$. The TRD T_r is created in advance before processing the other tuples in χ^p.

3.2 Processing a Single PMQT

Each PMQT χ^p (with associated TRD T_r and initial node (u, s_u)) can be evaluated by starting an invocation of Algorithm ***Insert*** with the node (u, s_u) to extend the partial match and find all satisfied paths in the streaming graph. Note that this algorithm is basically an instantiation of rule 2 and rule 3. Rule 1 is irrelevant since the root and the partial match are already determined.

Algorithm 1: *Insert*

Input: PMQT χ^p (with associated TRD T_r), node (u, s_u)

```
1  begin
2      initialize set M ← ∅;
3      foreach tuple τ = ((u, v, l), t) in χ^p do
4          foreach (s_u, s_v) s.t. δ(s_u, l) = s_v do
5              if (u, s_u) is in T_r then
6                  perform rule 2 to add node (v, s_v) and/or update (v, s_v).ts;
7                  if newly add (v, s_v) or change (v, s_v).ts then
8                      add node (v, s_v) to M;
9      foreach node (v, s_v) ∈ M do
10         Call Insert(χ^p, (v, s_v)) (i.e., rule 3);
```

The evaluation for a PMQT entails possibly multiple traversals for each tuple in the snapshot graphs in all the sliding windows that contain τ, to see if it can further extend an existing node of the TRD. Here we use an example to illustrate the evaluation process of the PMQTs using Algorithm ***Insert*** (Algorithm 1).

Example 3. *Consider the streaming RPQ evaluation under the same query conditions as in Example 1. Suppose we constrain the length of the partial match to $\ell en = 1$. Then, $\chi^{p_1=\{(e_1,0)\}}$ and $\chi^{p_2=\{(e_5,2)\}}$ are two valid PMQTs from the streaming graph S in Figure 2(a), which consist of edges e_0 to e_5 and e_2 to e_7, respectively. Next, we create the TRD T_a and invoke Algorithm 1 with initial node $(c, 1)$ for*

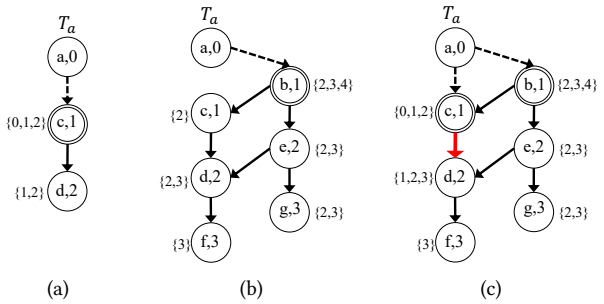


Fig. 3. (a) & (b) are the maintained TRD $T_{a,0}$ in PMQTs $\chi^{p_1=\{(e_1,0)\}}$ and $\chi^{p_2=\{(e_5,2)\}}$, respectively. The double circled nodes are the initial nodes. (c) is maintained TRD T_a when both PMQTs χ^{p_1} and χ^{p_2} are processed as a group.

χ^{p_1} and $(b,1)$ for χ^{p_2}. The maintained TRDs for these two PMQTs are shown in Figures 3(a) and 3(b), respectively.

A longer length threshold ℓen for the partial match length may lead to the generation of more PMQTs from the streaming graph, which can be particularly beneficial when the number of PMQTs is insufficient to fully utilize all the workers, or to process skewed data such as "hot" nodes in the streaming graph (which can be processed in parallel). However, this comes at the cost of increased processing time at the coordinator, as identifying and managing these partial matches requires additional efforts, which may become a bottleneck and significantly degrade performance. A proper ℓen can balance the processing cost in coordinator and the number of generated PMQTs to fully utilize the capabilities of all workers, which is verified in the experiments in Section 5.2.3.

Streaming Support. The above-described strategy assumes all tuples in a PMQT have arrived before being processed. However, we can also support the evaluation of the PMQTs in a streaming manner by invoking line 4 to 10 in Algorithm 1 for each newly arrived tuple τ', which is the same way as in [59]. For the sake of simplicity, in the sequel of this paper, we assume that a PMQT is built and processed when all its tuples have arrived, if not otherwise stated. We will explicitly discuss the streaming support when needed.

3.3 Processing Multiple PMQTs as a Group

Although TRDs with different root nodes can be maintained separately in parallel, PMQTs with the same root node may share some processing since different PMQTs may contain common tuples and their TRDs may have common subgraphs. When processing multiple PMQTs that maintain TRDs with the same root as a group, we first add all start nodes and end nodes to the TRD and merge the timestamps of the same node. Then, we take each PMQT (i.e., Algorithm 1) as a parallel process working on the same TRD.

Parallel execution of Algorithm 1 can be done with a *Readers-Writer Lock* on each TRD node to guarantee the update of line 6 be done sequentially. When there are two (or more) PMQTs that maintain the same TRD and update the timestamps of the same node (i.e., the same vertex and state), the subsequent update is done over the proceeding update (i.e., rule 2 is sequentially applied). If the subsequent update does not change the timestamp of a node in line 6, it means that the further extension of the node is already taken care of by a proceeding (parallel) process, and a recursive call in line 10 for the subsequent process is avoided. Such a shared TRD maintenance process is the same as done in [59].

Example 4. *Consider the streaming RPQ evaluation with the same query conditions as in Example 3. Both PMQTs χ^{p_1} and χ^{p_2} share the TRD T_a with the same root and both update the same nodes $(c, 1)$ and $(d, 2)$. Suppose the evaluation of χ^{p_1} first creates the node $(c, 1)$ and sets its timestamps $(c, 1).ts$ to set $\{0, 1, 2\}$. When the evaluation of χ^{p_2} attempts to update the timestamps of $(c, 1)$ after inserting node $(b, 1)$ (with edge e_0 in S), the calculated timestamps are $\{2\}$ (computed with $\{2, 3, 4\} \cap \{0, 1, 2\}$), which is a subset of the already updated timestamps $(c, 1).ts = \{0, 1, 2\}$. In this case, node $(c, 1)$ will not be added to set M in line 8, and the recursive call to Algorithm 1 for extending the path starting from $(c, 1)$ in the evaluation of χ^{p_2} is eliminated. The shared paths of χ^{p_1} and χ^{p_2} in T_a are marked in red in Figure 3(c). Note that the path marked in red may contain a large subgraph in practice, and the shared TRD maintenance process can save significant computation in such cases.*

Theorem 1. *The processing of multiple PMQTs generates all the correct query results.*

Proof: We need to prove that the decision of a given PMQT χ^{p_1} to not add a node to M (lines 7 to 8) due to an update by another PMQT χ^{p_2} does not affect the query results. As mentioned above, a PMQT χ^{p_1} avoids recursively calling Algorithm 1 when it encounters a node whose timestamp set ts is a subset of the timestamps ts' of the same node in the TRD already maintained by another PMQT χ^{p_2}. Note that both PMQTs use the same strategy to maintain TRDs, and the traversed tuples that may extend this node with rules 2 and 3 when processing PMQT χ^{p_1} are also a subset of these tuples when processing PMQT χ^{p_2}, since $ts \subseteq ts'$. Therefore, all the query results of either task will be included. This theorem can be extended to multiple PMQTs.

3.4 System Design

With the definition of *Partially Matched Query Task*, we now introduce DRPQ for the distributed streaming RPQ evaluation. The overview of the system is shown in Figure 4.

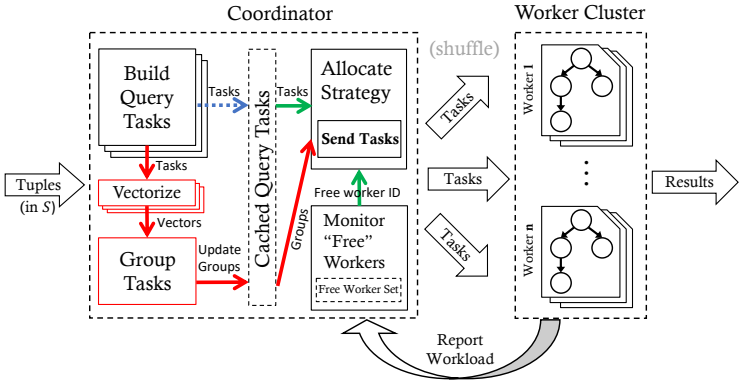


Fig. 4. The System Design of DRPQ

We adopt a *Coordinator-Workers* model to manage the servers in DRPQ. The coordinator continuously receives tuples from the streaming graph S, builds a PMQT for each founded partial match (with length threshold ℓen is given in advance) in S, and forwards them to workers based on an allocation strategy (one such strategy is introduced in Section 4). The PMQTs can be parallelly-built in the coordinator and send to workers. The workers receive PMQTs and process them as mentioned in Section 3.3. Note that we ignore the set semantics of the results in the distributed setting since [59] incorporates a post-processing thread to deduplicate endpoint pairs, ensuring set semantics in the single-worker case and a similar deduplication step can be integrated into

DRPQ as a lightweight post-processing phase after collecting results from all workers, if strict set semantics are required.

Theorem 2. *The query results obtained from DRPQ are correct and complete.*

Proof: As mentioned above, we build all PMQTs for the partial matches whose length $\ell en' = \ell en$ and all query tasks are correctly evaluated in workers (by Theorem 1). Besides, the other query instantiations with path whose length less than ℓen are also correctly founded in coordinator. We know that all query results in the streaming graph can be found in DRPQ.

3.4.1 Workload Balance. With the definition of PMQTs, it is easy for DRPQ to keep load balance among workers by monitoring the workload of the workers and dynamically sending PMQTs to workers. Note that the overlapping tuples in different PMQTs assigned to the same worker only need to be sent once to save communication costs, and such process can be accelerated by leveraging broadcasting. The workers and coordinator perform as follows:

- Each worker periodically checks its workload, which is represented by the number of PMQTs that have not yet been processed, and notifies the coordinator when its workload falls below a certain threshold.
- The coordinator monitors the notifications sent by workers, marks those workers as "free," and adds them to the "free worker set." Simultaneously, the coordinator collects the already-built PMQTs and places them in the "Cached Query Tasks List," as shown by the dotted line in Figure 4. Both the free worker set and the Cached Query Tasks List are structures maintained by the coordinator. When the free worker set is not empty, the coordinator selects a certain number of PMQTs from the Cached Query Tasks List and assigns them to a worker chosen from the free worker set (the worker is removed from the set after selection), as indicated by the green arrows in Figure 4.

The above process achieves balanced workload among the workers (shown as the good performance in experiments), with the assumption that each PMQT takes about the same time to evaluate. In reality, this assumption may not be true. In that case, a more sophisticated monitoring may be done to determine if a worker is free, but is beyond the scope of this paper.

3.4.2 Discussion. DRPQ is able to handle the out-of-order arrivals of the tuples by constructing and evaluating PMQTs in a streaming manner. The only requirement is to track PMQTs assigned to each worker and route incoming tuples based on their timestamps to the appropriate workers for streaming processing. In addition, the dirty garbage collection strategy for TRD (mentioned in Section 2.2) can be performed on each worker by synchronizing the timestamps of the tuples the coordinator last sends. Besides, DRPQ also supports explicit deletions of the tuples by sending the deletion tuples to the associated PMQTs and processing them in the same way in [59].

3.4.3 Cost Comparison of Distributed Strategies. DRPQ offers significant advantages in distributed streaming graph processing by achieving an optimal balance between computation, communication, and load balancing. Unlike the Splitting Query and Window Partitioning strategies, which suffer from excessive computational overhead due to costly joins or duplicate processing across multiple sliding windows, DRPQ effectively minimizes unnecessary computations. Its duplicate computation cost is substantially lower than the expensive multi-join (or merge) operations in Splitting Query strategy and the repeated processing in Window Partitioning. Furthermore, DRPQ significantly reduces inter-worker communication compared to Graph Partitioning strategy, which incurs high messaging overhead due to frequent cross-worker path traversals. Beyond these efficiency gains, DRPQ also excels in load balancing by ensuring an even workload distribution, thereby preventing the bottlenecks and imbalances that hinder other strategies. These combined advantages make

DRPQ a highly efficient and scalable solution for large-scale distributed streaming systems. A detailed experimental comparison is presented in Sections 5.2.1 and 5.2.2.

4 Group-based Allocation Strategy

In this section, we focus on the coordinator's allocation strategy, which customizes PMQTs to improve shared TRD maintenance (see Section 3.3). We first analyze key features of PMQTs (Section 4.1), then introduce a vectorization-based similarity measure and its approximation (Section 4.2), and finally present a grouping method based on vector similarity (Section 4.3).

4.1 Feature Vectorization of PMQTs

Recall the discussion in Section 3.3, PMQTs can share maintenance processes in workers if their associated TRDs meet three key conditions: **(a)** they have the same root node, **(b)** they contain common nodes (i.e., nodes with the same vertex and state), and **(c)** the timestamps of these common nodes exhibit a containment relationship (one set containing the other). For example, in Figure 3, node $(c, 1)$ satisfies these conditions, while $(d, 2)$ does not. From this, we derive two key insights: (1) PMQT sharing is only relevant when their TRDs have the same root node, and (2) the more nodes that satisfy conditions (b) & (c), the greater the potential for shared processing.

To quickly judge how much PMQTs may have shared processes, we first represent each PMQT as an $n \times k$ integer vector. Here, n denotes the total number of possible nodes (u, s_u) in any TRD of PMQTs, with each node assigned a unique integer from 1 to n, and k represents all possible timestamps on nodes. Conceptually, this vector can be viewed as an n-row, k-column binary matrix, where each entry is set to 1 if the corresponding node and timestamp exist in the TRD, and 0 otherwise. Then, based on these generated vectors, we estimate the possibility of shared TRD maintenance by computing a similarity measure between PMQTs with the same root node, which is represented as the overlap length of timestamps on common nodes between two PMQTs.

Definition 10. *Similarity: For two PMQTs χ^{p_1} and χ^{p_2} with associated TRD T_u and T_v, respectively, the similarity $Sim(\chi^{p_1}, \chi^{p_2})$ between them is calculated as follows:*

$$Sim(\chi^{p_1}, \chi^{p_2}) = \begin{cases} vct_i * vct_j, & if\, u = v \\ 0, & otherwise \end{cases} \tag{1}$$

where vct_i and vct_j are the vectors associated with χ^{p_1} and χ^{p_2}, respectively.

Given two PMQTs χ^{p_1} and χ^{p_2}, $Sim(\chi^{p_1}, \chi^{p_2})$ is basically the dot product of the associated vectors, which represents the number of overlapping timestamps on the common nodes of their TRDs. Note that $Sim(\chi^{p_1}, \chi^{p_2}) \in [0, n * k]$. Also note that the larger $Sim(\chi^{p_1}, \chi^{p_2})$ is, the more overlapped timestamps on the common nodes of the TRDs, and we say the more similar they are.

4.2 Approximated Vectorization

The above theoretical vectorization method is obviously impractical since both n and k are extremely large. Moreover, it is impossible to have the complete TRD of a PMQT unless we have already constructed the TRD for the PMQT (but this is exactly the process that the workers need to perform). We thus propose an approximated vectorization of the PMQTs based on their own partial matches.

First, we merge the nodes into m different buckets and add the corresponding k elements of the merged nodes together, i.e., add the corresponding timestamps of all the nodes mapped to the same bucket. Hence, the compressed vector may still be large in theory (i.e., $m * k$), but the computation of vector similarity (see below) can take an efficient form. Indeed, the k elements of each bucket are very sparse. Hence, we only need to remember the non-zero elements of the vector. **Second**, we adopt a heuristic method that only considers the part of a TRD for each PMQT, restricting

nodes to those within at most h hops from the initial node (u, s_u). This partial TRD can be quickly constructed by the coordinator.

The above approximation sacrifices some accuracy. The larger the hop h, the more nodes are sampled and the more time cost it takes to obtain such a (partial) TRD, but the higher possibility the coordinator knows which query tasks can share processing. Besides, the nodes compressed into the same bucket are taken as the same. We may control the size of m and hop h to balance the grouping quality and cost. We will discuss this trade-off in the experiments.

In summary, we efficiently vectorize a PMQT χ^p with an associated TRD T_x and initial node (u, s_u) into a $m \times k$-dimensional vector vct using the following steps: (1) Construct a TRD with all simple paths in the product graph from the initial node (u, s_u) at most length h and collect all the nodes along with their timestamps. (2) Adopt an assignment function (using hash function in this paper) to quickly assign these nodes to m buckets. (3) Each bucket maintains a compressed subvector vct' of length k (but compressed using their sparsity), where the i-th dimension $vct'[i]$ represents the number of nodes in that bucket with corresponding timestamps (set to 0 if none exist). Here we use an example to illustrate the process of vectorizing a PMQT.

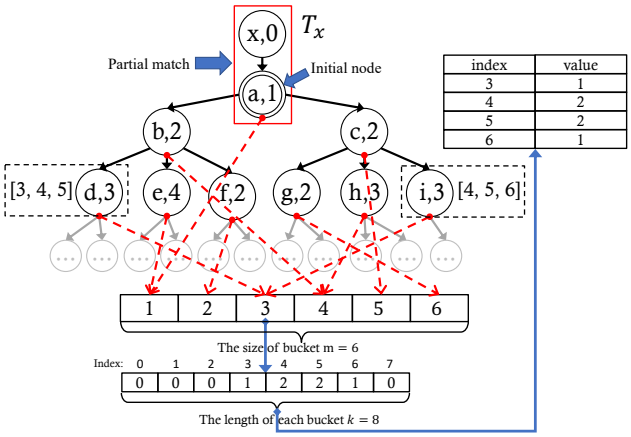


Fig. 5. Vectorize a PMQT

Example 5. *Figure 5 shows an instance to vectorize a pre-given PMQT with associated TRD T_x and initial node $(a, 1)$. We set the hop to $h = 2$, the number of buckets to $m = 6$, and the total possible timestamps to $k = 8$. First, we construct the hop-2 TRD T_x for the PMQT, which is shown in this figure with nodes and edges that are marked in black. Next, we assign these obtained nodes into 6 buckets based on the assignment function, as indicated by the red dotted arrows. Notably, both nodes $(d, 3)$ and $(i, 3)$ are mapped to the same bucket with ID 3, and the subvector of this bucket is shown underneath where each the number indicating the count of the particular timestamp appearing in $(d, 3).ts = \{3, 4, 5\}$ and $(i, 3).ts = \{4, 5, 6\}$. An efficient representation of this subvector is shown in the table in the upper-right corner.*

The Eq. 1 can be adopted when vectors are approximated with the method discussed earlier. Since we consider all nodes that are compressed into the same bucket to be the "same", for each node in vct_1, we need to calculate the overlapped timestamps with all the "same" nodes in vct_2. That is, assuming that x nodes in vct_1 with corresponding subvectors $n_1, \cdots, n_x$ and y nodes in vct_2 with corresponding subvectors $m_1, \cdots, m_j$ are the mapped to the same bucket, then the similarity

of the two buckets is calculated by:

$$\sum_{i=1}^{x}\sum_{j=1}^{y} n_i * m_j = \sum_{i=1}^{x} n_i * \sum_{j=1}^{y} m_j \tag{2}$$

Thus, we can add all the subvertors of the nodes in each bucket and each PMQT is then represented by an approximated vector in which each bucket (i.e., merged node) has an associated subvector. Then the similarity between two PMQTs when their feature vectors are approximated can thus be calculated in the same way as Eq. 1.

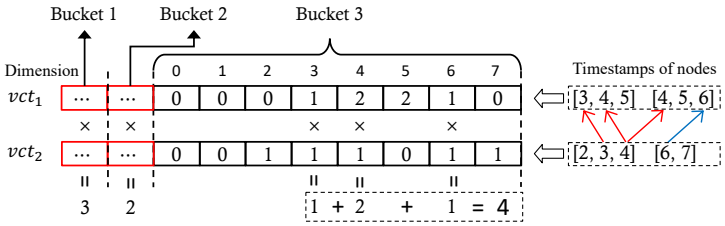


Fig. 6. Calculate similarity with two vectors vct_1 and vct_2

We now use an example to illustrate calculating similarity between different approximate vectors.

Example 6. *The example of Figure 6 illustrates the process of calculating the approximated similarity of two query tasks with the associated vectors vct_1 and vct_2 with bucket size $m = 3$ and $k = 8$. For the bucket ID $= 3$, assume this bucket in vector vct_1 consists of two nodes with timestamps $\{3, 4, 5\}$ and $\{4, 5, 6\}$ and in vector vct_2 consists of two nodes with timestamps $\{2, 3, 4\}$ and $\{6, 7\}$. Their corresponding approximate subvectors are shown in the figure. The similarity in this bucket is calculated as the dot product of their corresponding subvectors, which is 4. Roughly, this result 4 gives the number of overlapping timestamps in this bucket between the two PMQTs. The other buckets 1 & 2 are calculated in the same way. The similarity of these two PMQTs is $3 + 2 + 4 = 9$.*

4.3 PMQT Grouping

The coordinator groups PMQTs that are likely to share their TRD maintenance process before assigning them to workers. For each root node, it maintains a set of groups, where each group g contains query tasks with the same root, formed based on the similarity measure described in Section 4.2. Each group g is represented by an $m \times k$-dimensional vector, obtained by summing all member vectors in the group. The similarity is calculated between χ^{p_1} and a group g by replacing χ^{p_2} with g and u_2 with the root of g.

For a new PMQT χ^p, the coordinator calls Algorithm ***Grouping*** (Algorithm 2) to add this PMQT to the maintained group set (create a new group if needed). It calculates the similarity between the given PMQT and each group g (Line 4 in Algorithm 2). Then, if the maximum similarity calculated above is above a certain pre-defined "*similarly threshold*", add the given PMQT to the most similar group g in lines 6 and 7 in Algorithm 2. Otherwise, i.e., the PMQT is not similar enough to any of the groups, a new group is created with the given PMQT in lines 9 to 10 in Algorithm 2.

4.3.1 Grouping-Based Coordinator Design. The coordinator efficiently manages PMQT assignment using a vectorization and grouping strategy, as highlighted in red in Figure 4. Each pre-built PMQT is first vectorized in parallel within the "Vectorize" module, after which Algorithm 2 is applied sequentially to group PMQTs. These grouped query tasks are stored in the "Cached Query Tasks"

Algorithm 2: *Grouping*

Maintained: group set (of a given root node r)
Input: PMQT χ^p with associated TRD T_r and vector vct

```
1  begin
2      initial similarity map M;
3      foreach group g in group set do
4          calculate similarity M(g) ← Sim(vec, g);
5      if max value {M} > a pre-given similarity threshold then
6          find the group g with the max value M(g);
7          add vct to group g;
8      else
9          create a new empty group g and add it to the group set;
10         add vct to group g;
```

for dynamic assignment, and the coordinator prioritizes assigning PMQTs from the same group to a single worker. Upon detecting an available worker, it selects and dispatches a batch of PMQTs from the cache. The coordinator continuously updates these groups by adding new PMQTs through Algorithm 2 and removing dispatched tasks. It also tracks the groups previously assigned to each worker, ensuring that newly assigned tasks belong to groups the worker has already processed when possible. If no PMQTs remain in those groups, the coordinator assigns tasks from a new group. Additionally, to maintain load balance, the coordinator dynamically split existing groups and distribute them among multiple free workers when necessary.

4.3.2 Discussion. We note that both PMQT generation and grouping at the coordinator are lightweight operations in most scenarios, and can be parallelized using multi-threading. However, these behaviors are highly context-sensitive, influenced by both static parameters (e.g., partial match length and grouping granularity) and dynamic workload characteristics such as stream skew, query selectivity, and temporal burstiness. In certain settings, especially when the correlation among PMQTs is weak or when the data stream exhibits bursty behavior, the benefit of reducing redundant computation may be offset by the coordination overhead introduced by grouping. While we currently adopt a static grouping policy and fixed parameter settings (e.g., bucket size, hop, and similarity threshold), we acknowledge that adaptive strategies—capable of dynamically tuning grouping decisions and parameters in response to observed workload patterns—can offer better performance. We leave the design of such adaptive strategies and runtime plan switching mechanisms as promising directions for future work.

5 Experimental Evaluation

In this section, we experimentally evaluate the performance of the proposed methods. We begin by introducing the system implementation and experimental settings in Section 5.1. Then, in Section 5.2, we compare DRPQ with other distributed strategies and provide a detailed evaluation and analysis of the grouping strategy.

Here we give a brief highlight of our experimental results:

- DRPQ demonstrates excellent throughput and scalability compared to the other strategies when evaluating streaming RPQs in a distributed manner. Specifically, the throughput increases nearly linearly as the number of workers grows.

- The grouping strategy in DRPQ significantly enhances performance, improving throughput by almost twofold in most query scenarios and conditions.

5.1 Experimental Setup and Implementation

Experimental Setting: All experiments were conducted on a cluster of 9 machines (1 coordinator and 8 workers), each equipped with an Intel Xeon (Cascade Lake, 2.30 GHz), 32 physical cores, and 256 GB RAM, interconnected via a 10 Gbps network.

Implementation of Compared Approaches: We implemented all the aforementioned distributed strategies in Java 8. Streaming graph tuples are generated by continuously reading from a file, and results produced by workers are collected by an external machine (not included in the system). Communication between the coordinator and workers is implemented via asynchronous MPI-based message passing (MPJ v0.43 [30]), with each worker maintaining a dedicated receiving thread. All approaches being compared share the same core TRD-based evaluation engine [59] with the TRD serving as the unified internal structure for incremental RPQ processing across all workers, and differ only in how tuples are distributed to workers and how each worker organizes its computation and synchronization logic accordingly. Following are the implementation details of the tested approaches in experiments.

- **DRPQ:** The coordinator incrementally builds partially matched query tasks (PMQTs) based on observed partial paths in the stream and sends them to workers. Each worker runs the TRD-based evaluation algorithm starting from the pre-initialized partial match encoded in the PMQT. Each worker independently maintains local TRDs and outputs matching results. We implement two variants: *DRPQ (random)* randomly assigns PMQTs to workers, while *DRPQ (grouping)* applies the grouping strategy described in Section 4. Internal modules like "Vectorize" and "GroupTasks" are implemented as Java thread pools. The partial match length threshold is set to $\ell en = 1$ by default, and the grouping parameters are: bucket size $m = 100$, similarity threshold $st = 10$, and hop $h = 2$. A worker reports itself as "free" when its PMQT queue length drops below 100, and the coordinator assigns a batch of 1,000 PMQTs.

- **Split Query:** This approach follows the method in [46], where the regular expression is decomposed into a binary join tree (query plan). Each subexpression is evaluated independently by a worker on the entire stream. The coordinator receives intermediate results and performs multi-way joins to produce final answers. We reuse the same TRD engine to evaluate each subquery. Intermediate results are buffered and joined in memory using hash-join structures.

- **Window Partition:** In this strategy, the entire query is evaluated repeatedly over disjoint or overlapping windows. The coordinator assigns each sliding window instance to a different worker. The worker builds a local TRD for its assigned window and evaluates the full query independently. The implementation shares the same evaluation engine but resets TRD states between windows. Tuples are routed according to their timestamps.

- **Graph Partition:** We implemented four edge-partitioning schemes—Greedy, HDRF, DBH, and Grid [1]—to divide the graph. Each tuple is routed to workers based on the partition of its nodes. Since query paths can cross partitions, workers exchange intermediate TRD states for edges that connect across workers. Each worker maintains TRDs locally but supports TRD propagation via MPI. We select the best-performing partitioner per run.

Note that all three baseline strategies can be viewed as naive distributed extensions of the TRD-based evaluation algorithm in [59], where distribution is achieved through static decomposition of the query, the graph, or the time domain, respectively.

Workloads & Datasets: The queries (workloads) used in our experiments are regular expressions listed in Table 1. Specifically, Q_{1-8} represent common recursive query patterns observed in real-world applications [11]. Q_9 is a manually constructed query designed to evaluate system

Table 1. The regular expressions used in our workloads.

Case	Query	Case	Query
Q_1	$a^* \circ b? \circ c$	Q_4	$a^* \mid b^*$
Q_2	$a \circ (b \circ c)^*$	Q_5	$a \circ b^* \circ c^*$
Q_3	$a^* \circ b^*$	Q_6	$a \circ (b^* \mid c)$
Q_7	$(a_1 + a_2 + a_3 + \cdots + a_k)^*$		
Q_8	$(a_1 + a_2 + a_3 + \cdots + a_k) \circ b^*$		
Q_9	$a_1^+ \circ (a_2 \mid a_3^*) \circ a_4 \circ a_5? \circ a_6^*$		

robustness under more complex query structures. Note that the query results for Q_3, Q_4, and Q_7 must include at least one label, as we require matching paths to be non-empty (specified in Definition 4). We evaluate these queries on two real graphs:

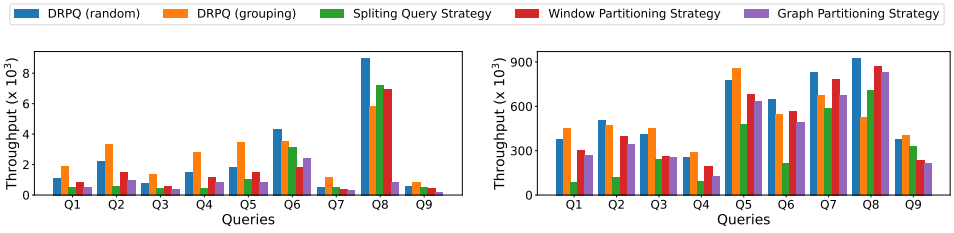
- **Stack Overflow** (SO) is a network of Question & Answer which consists of approximately 2.6 million users (vertices) and 74 million interactions (edges) spanning six years [44] up to now. Interactions are classified into three types based on the edge label l in each tuple (e, t), where $e = (u, v, l)$: $(l = a)$ denotes that user u answered a question by user v at time t; $(l = b)$ means that u commented on v's answer; and $(l = c)$ indicates that u commented on v's question. The timestamp t records when the interaction occurred. Unless otherwise specified, we set the sliding window size $|W|$ to 1 week ($|W| = 7$) and the slide length $Step(W)$ to 1 day.

- **Yago4** is the latest version of the YAGO knowledge base derived from Wikidata [48]. It contains over 50 million entities (vertices) and 2 billion facts (edges), including a rich schema with 157 relation labels. To simulate sliding windows on the Yago4 RDF graph, we assign a monotonically non-decreasing timestamp to each RDF triple at a fixed rate. Each window in Yago4 contains approximately 10 million edges (|W| = 10M) and slides every 100K edges. For YAGO, the queries in Table 1 are instantiated using real relation labels from the dataset. Labels are selected manually to match the query pattern and ensure sufficient support. While our focus is on structural diversity for system evaluation, benchmarks like [3] with real query logs are valuable for future extension.

It is worth noting that both SO and YAGO datasets naturally exhibit skewed characteristics, including high-degree nodes, uneven edge distributions, and varied path lengths. These inherent properties provide a realistic basis for assessing the system's robustness to data and workload skew in practical settings.

Performance Metric: We use *throughput* as the primary performance metric, defined as the number of streaming tuples the system can process per second while maintaining correct and up-to-date query results. We will briefly discuss the tail latency as well. Recall that all components in the grouping operation can be executed as a pipeline, the grouping strategy and query processing can proceed in parallel, and task distribution incurs negligible overhead. The observed throughput gains are sufficient to contextualize the overhead introduced by DRPQ and the grouping strategy. We report average throughput over a 6-minute warm-cache run to reflect steady-state system performance.

It is worth noting that we also measured the necessary communication overhead between the coordinator and workers, including PMQT dispatching and control messages in DRPQ. This cost remains consistently low—under 3% of total processing time—thanks to asynchronous message passing and dedicated receiving threads. Therefore, we consider the communication overhead of DRPQ negligible and omit it from our experimental breakdowns.

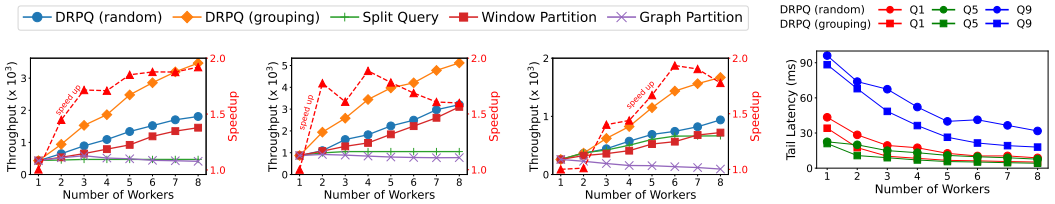


(a) Stack Overflow (b) Yago4

Fig. 7. Evaluate throughput with different distributed strategies and different query cases on two datasets.

5.2 Experimental Results

5.2.1 Performance on Different Queries and Datasets. We first evaluate the performance of distributed strategies using four workers across all queries in Table 1 on both datasets, with results shown in Figure 7. It is evident that each strategy has its strengths across different queries and datasets. However, the throughput of DRPQ (random) and DRPQ (grouping) consistently outperforms the other distributed strategies for all queries, with a performance more than twice that of the other strategies in most cases. Specifically, DRPQ (random) and DRPQ (grouping) show significantly higher improvements under conditions of lower throughput (i.e., heavier workloads) on both datasets, with gains reaching up to 3×. Furthermore, compared to DRPQ (random), DRPQ (grouping) achieves considerable throughput gains on most queries in SO, including structurally more complex one Q_9. The grouping strategy delivers greater improvements in scenarios of lower throughput (e.g., a 2.3-fold increase in throughput for Q_7). This improvement can be attributed to the substantially larger size of the TRDs under heavy workloads, which enables more shared processes across different PMQTs. An exception is observed for $Q_{6,8}$ in SO and $Q_{2,6-8}$ in YAGO, where the time costs associated with vectorizing and grouping significantly exceed those for processing, creating a performance bottleneck.



(a) (b) (c) (d)

Fig. 8. (a), (b), and (c) show the throughput of different strategies with Q_1, Q_5, and Q_9, respectively. (d) shows the tail latency of DRPQ (both random and grouping strategies) using different numbers of workers.

5.2.2 Scalability Analysis. We use query Q_1, Q_5, and Q_9 to evaluate the scalability of the distributed strategies by monitoring throughput across different numbers of workers. All other settings remain as described in 5.2.1, and the results are shown in Figure 8(a), 8(b), and 8(c). The results for one worker correspond to the single-worker execution of DRPQ, serving as a baseline to measure distributed speedup. All the results show a similar trend that, as the number of workers increases, only the DRPQ strategy (both grouping and random) and Window Partitioning strategy demonstrate a **steady, linear** increase in throughput. In contrast, the Splitting Query strategy is constrained by

the number of split parts of the query, and the Graph Partitioning strategy experiences decreasing throughput due to a sharp rise in communication overhead. The throughput of DRPQ is consistently higher than that of the Window Partitioning strategy, highlighting the superior scalability of our proposed DRPQ system. Additionally, we assess the scalability of the grouping strategy by monitoring the speed-up ratio (calculated as DRPQ (grouping) / DRPQ (random)) across different worker counts, shown by the red dashed line in the figure. As the number of workers increases, DRPQ (grouping) demonstrates a significant throughput improvement over DRPQ (random), nearly doubling the throughput when 8 workers are used. Compared to all other distributed strategies, DRPQ (grouping) exhibits exceptional scalability, doubling its throughput as the number of workers increases from 2 to 4 to 8.

We also evaluate the scalability of DRPQ by monitoring the tail (99th percentile) latency under various number of workers. The results in Figure 8(d) show that as the number of workers increases, the tail latency for both DRPQ (random) and DRPQ (grouping) decreases consistently across all query types (Q_1, Q_5, Q_9). Notably, DRPQ (grouping) achieves lower tail latency than DRPQ (random) in all configurations, with more significant reductions observed in higher workloads such as Q_9.

Table 2. Throughput under different partial match length ℓen

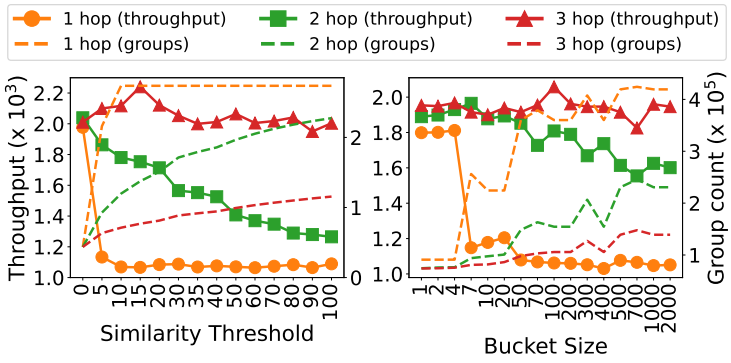
ℓen	**1**	**2**	**3**	**4**	**10**	**100**	**1000**
DRPQ (random)	182	245	368	221	111	105	107
DRPQ (grouping)	195	429	533	242	81	102	101

5.2.3 Threshold of Partial Match Length. To explore the effect of the partial match length threshold ℓen under a limited number of workers, we selected 1M tuples from the SO dataset and controlled the number of generated PMQTs as follows: (1) We introduced two new labels, d and e, which together form several paths (about 0.01% of tuples in SO) with the label sequence $d \circ e$; (2) We identified several hot nodes (about 0.001% of nodes in SO) that connect to the source vertex of all tuples labeled a via e. The sliding window size was set to 90 (three months), and the query was $Q' = d \circ e \circ Q_1$. Other experimental settings followed Section 5.2.1; results appear in Table 2. As discussed in Section 3.4, the optimal threshold in this setting is $\ell en = 3$, which balances generating sufficient PMQTs and limiting coordinator overhead. For $\ell en < 3$, the bottleneck is too few PMQTs; for $\ell en > 3$, the coordinator incurs excessive cost finding partial matches. Notably, throughput remains unchanged when ℓen exceeds 100, as most partial matches fail the length threshold and the system effectively degenerates to single-coordinator processing, diminishing distributed benefits. A suitable ℓen can typically be found by conducting preliminary tests on sampled data.

5.2.4 Parameters in Grouping Strategy. We now evaluate the influence of the parameters (bucket size m, hop h, and $threshold$ in Algorithm 2) in the grouping strategy to the grouping quality. Note that these three parameters interact and collectively influence the grouping quality, we thus conduct several experiments on SO and query Q_1 to evaluate the covariance impact among them.

Figure 9(a) presents the throughput and number of generated groups under varying hop h and similarity threshold st, with bucket size fixed at $m = 100$. We exclude $h = 4$ because its grouping cost far exceeds processing cost, introducing a new performance bottleneck.

Notably, for $h = 1$ and $h = 2$, throughput steadily decreases as st increases. Specifically, at $h = 1$, throughput nearly halves at $st = 1$ and then stabilizes for $st \geq 10$, closely resembling the DRPQ (random) strategy (see black dashed lines in the figure). For $h = 2$, throughput declines more gradually with increasing st. In contrast, for $h = 3$, throughput initially rises as st increases, peaking at $st = 15$ (the "turning point"), after which it declines.



(a) (b)

Fig. 9. The throughput with (a) different hops & similarity thresholds and (b) different hops & bucket size.

These trends can be explained as follows. When $h = 3$, increasing st allows more PMQTs within a group to share TRD maintenance, boosting throughput. Beyond the turning point, stricter similarity constraints split previously grouped PMQTs, resulting in throughput decline. For $h = 2$, fewer sampled nodes mean the system reaches or exceeds the turning point sooner, causing a slower but steady decline as st increases. The same logic applies to $h = 1$, which behaves like DRPQ (random) due to the less sampled nodes.

The number of groups generated is shown by the dashed lines in Figure 9(a), and is almost exactly inversely proportional to throughput: more groups correspond to more random-like allocation and lower TRD sharing. At $st = 0$, group counts are similar across all hops since a loose threshold easily satisfies the overlap condition, grouping more PMQTs together. As st increases, grouping becomes stricter and the number of groups rises across all hops. Higher h values (i.e., more sampled nodes) result in fewer groups for the same st, as more PMQTs can satisfy the grouping requirements.

Next, we evaluate the impact of bucket size m and hop h on performance with similarity threshold $st = 5$, as shown in Figure 9(b). For $h = 1$, throughput steadily decreases as m increases, with a sharp drop when m rises from 4 to 7, after which it stabilizes. For $h = 2$, throughput increases with m up to a peak at $m = 7$, and for $h = 3$, the peak occurs at $m = 100$. Notably, the turning point for $h = 3$ yields higher throughput than that for $h = 2$.

These results are consistent with previous findings. Increasing m makes it easier to satisfy condition (b) in Section 4, allowing more PMQTs in a group to share TRD maintenance and increasing throughput—up to the turning point. Beyond this, the fixed number of sampled nodes per hop leads to fewer nodes in each bucket, making it harder to meet the similarity threshold, and thus previously grouped PMQTs are split into separate groups, reducing throughput. The turning point appears earlier for $h = 2$ ($m = 7$) than for $h = 3$ ($m = 100$), as the latter samples more nodes and thus better supports feature-based grouping.

The number of groups under different settings is indicated by the dashed lines in Figure 9(b). Unlike Figure 9(a), bucket size has a stronger effect on group count, as it directly determines how sampled nodes are distributed. This results in more significant variation in group formation and consequently influences throughput.

We now fix the hop $h = 2$ and evaluate the impact of bucket size m and similarity threshold st on performance, as shown in Figure 10. The turning point appears when $m \leq 20$, with higher m values corresponding to lower st thresholds needed to reach the turning point (e.g., $st = 80$ at $m = 1$, $st = 60$ at $m = 2$, $st = 35$ at $m = 4$). This is because larger m means fewer nodes per bucket,

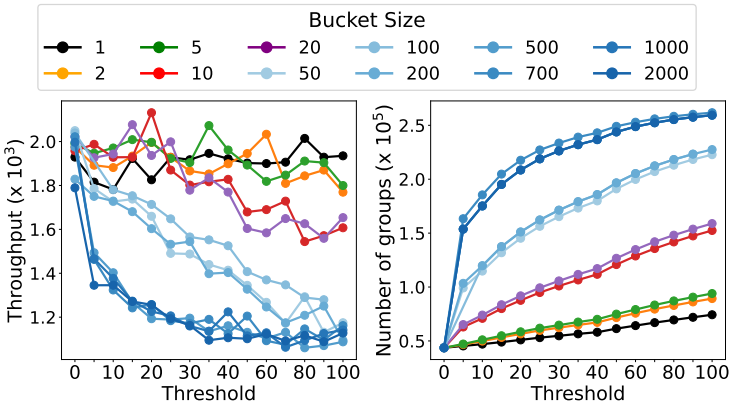


Fig. 10. The throughput and number of groups with different similarity thresholds and bucket size.

so a lower st is sufficient to distinguish which PMQTs can share TRD processes. When $m \geq 50$, the turning point disappears and throughput declines more rapidly, as each bucket contains too few nodes to satisfy the similarity requirement; equivalently, st in this range already exceeds the turning point. Notably, higher m values achieve higher throughput at their turning point (e.g., throughput for $m = 10$, $st = 20$ is greater than for $m = 4$, $st = 35$ or $m = 2$, $st = 60$), since larger buckets facilitate finer differentiation and better process sharing among PMQTs.

In summary, for a given query, a higher hop h is generally preferable because it samples more nodes for vector generation, as long as grouping overhead does not become a bottleneck. The optimal choice of m and st depends on the number of sampled nodes: with fewer nodes, smaller m and higher st are preferable, while with more nodes, larger m and lower st are more effective for grouping and throughput. Since optimal parameter settings depend on both dataset and query complexity, we recommend preliminary experiments on sample data to tune parameters prior to deployment.

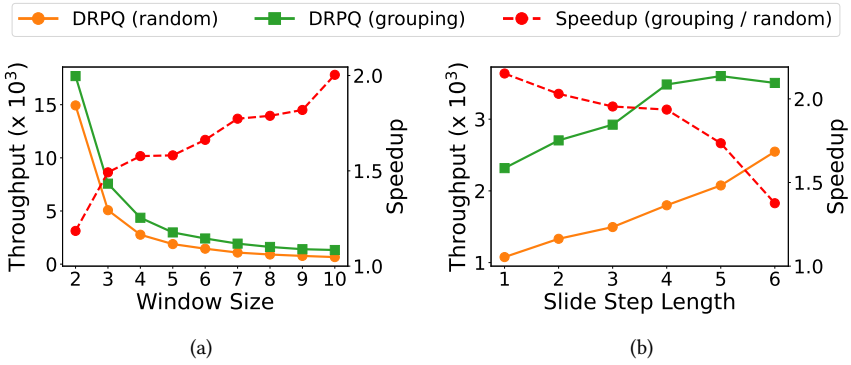


Fig. 11. The throughput of adopting grouping strategy with (a) different sliding windows and (b) sliding step length.

5.2.5 Sensitivity Analysis. We evaluated the sensitivity of our grouping strategy by measuring the throughput of DRPQ (random) and DRPQ (grouping) under different sliding window sizes

($|W|$) and sliding step lengths ($Step(W)$) for query Q_1. The results are presented in Figures 11(a) and 11(b). As $|W|$ increases, the throughput of both methods declines due to the heavier query workload within each window. However, the speed-up achieved by the grouping strategy over DRPQ (random) (depicted by the red line) increases, since larger windows result in larger TRDs and each PMQT contains more tuples. This leads to more opportunities for TRD maintenance to be shared across different PMQTs, allowing the grouping strategy to achieve higher relative throughput as $|W|$ increases.

In contrast, as $Step(W)$ increases, the throughput of both methods improves because a larger $Step(W)$ reduces the number of tuples within certain PMQTs (see Definition 9). However, our experiments show that when $Step(W) = 4$ (with $|W| = 7$), the throughput of the grouping strategy stops increasing, as the overhead of grouping surpasses the processing time on workers, creating a new bottleneck. Additionally, as $Step(W)$ increases, the speed-up ratio of the grouping strategy gradually decreases, since there are fewer opportunities for sharing computation across PMQTs.

6 Related Work

Regular Path Queries (RPQs): RPQs are widely used in graph querying [4, 5, 20]. Early RPQ evaluation methods can be categorized as navigational or relational. Navigational methods include rare label splitting [34, 42], Brzozowski derivatives [43], bidirectional random walks [55], and partial answers [56]. [8] combines the Glushkov automaton [23] with a ring index [7]. Relational methods focus on recursive queries, such as recursive SQL [19, 58] and relational algebra [22, 29]. Recent advances employ hardware/software acceleration, like FPGA-based parallel RPQ evaluation [39] or JIT compilation for direct graph evaluation [52]. Pang et al. [47] optimize query processing under memory constraints by exploring shared subqueries and materialized views.

Most prior works focuses on static graphs. [57] study reachability queries that aim to identify time-respecting paths in temporal graphs. However, their approach targets static settings with offline indexing, and does not support regular path queries or streaming evaluation. [45] introduces persistent RPQ evaluation over streaming graphs, later extended in [46] by integrating RPQs with graph pattern matching and proposing a streaming graph algebra. Gou et al. [25] reduce memory and update costs using shared spanning trees to materialize intermediate results. [59] is the first to support parallel processing over multiple sliding windows, avoiding expiration (see Section 2). However, no distributed solution is offered.

Streaming Graph Processing Systems: Streaming RDF processing has been explored in various contexts, including Linked Data Notification [12], RDF stream publishing [6], SPARQL extensions for reasoning and event matching [33], and persistent query systems like C-SPARQL [9], CQELS [36], and SPARQL$_{stream}$ [13]. However, these systems do not address RPQs specifically [60]. Streaming graph algorithms are divided into two categories: (1) processing large static graphs in a streaming manner, focusing on graph compression [2, 37] and partitioning [14, 49]; and (2) maintaining streaming graphs and dynamically updating outputs, including persistent queries for subgraph matching [17, 31], triangle counting [26, 51], cycle detection [50], and dynamic indexing for connectivity queries [15, 61]. Our work falls into the second category, focusing on regular expression queries. It extends prior efforts [25, 45, 46, 59] by enabling parallel RPQ evaluation in distributed systems, overcoming limitations like shared spanning tree structures and optimizing TRDs for high-performance processing. The shared TRD maintenance is conceptually related to the multi-source BFS optimization in [53], which merges traversals from multiple roots in static graphs. However, DRPQ differs fundamentally in that it targets streaming, labeled graphs, supports RPQs guided by automata, and merges across PMQTs from the same root under sliding window constraints. DRPQ also resembles that of [10] and [21], but with temporal constraints: while both works operate on static graphs and [21] retains all paths, TRDs in DRPQ is designed for streaming scenario and expire

as window sliding. Though we currently focus on node pairs, TRDs preserve sufficient information to reconstruct paths if needed.

A more detailed discussion of distributed strategies for streaming evaluation is provided in Section 1.

7 Conclusion and Future Work

In this paper, we proposed DRPQ, a novel approach for efficient distributed evaluation of streaming RPQs. We introduced the PMQT as a flexible unit of work and developed a grouping strategy to merge PMQTs that can share computation, thereby reducing redundancy. Experimental results show that DRPQ consistently outperforms existing distributed strategies across diverse queries, achieving near-linear throughput scaling with the number of workers. The grouping strategy further boosts throughput by nearly twofold in most cases. Future work mainly focuses on extending the concept of DRPQ to other streaming graph query applications and exploring adaptive runtime partition and grouping strategies at the coordinator, including dynamic adjustment of grouping parameters and condition-based switching between grouping and random modes.

Acknowledgments

This work was supported by NSFC under grants U24A20232 and 62272106. The corresponding author of this paper is X. Sean Wang.

References

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603.

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. doi:10.1145/2213556.2213560

[3] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. Wdbench: A wikidata graph query benchmark. In *International Semantic Web Conference*. Springer, 714–731.

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

[5] 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.

[6] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*. 635–644.

[7] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-case optimal graph joins in almost no space. In *Proceedings of the 2021 International Conference on Management of Data*. 102–114.

[8] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time-and space-efficient regular path queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3091–3105.

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2009. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th international conference on World wide web*. 1061–1062.

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

[11] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. *the web conference* (2019).

[12] Jean-Paul Calbimonte. 2017. Linked data notifications for rdf streams. In *Proceedings of the Web Stream Processing workshop (WSP 2017) and the 2nd International Workshop on Ontology Modularity, Contextuality, and Evolution (WOMoCoE 2017) co-located with 16th International Semantic Web Conference (ISWC 2017)*. 22 October 2017.

[13] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. 2010. Enabling ontology-based access to streaming data sources. In *The Semantic Web–ISWC 2010: 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I 9*. Springer, 96–111.

[14] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2018. PowerLyra. *ACM Transactions on Parallel Computing* (Sep 2018), 1–39. doi:10.1145/3298989

[15] Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu. [n. d.]. DLCR: Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs. ([n. d.]).

[16] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.

[17] Sutanay Choudhury, LawrenceB. Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. *Extending Database Technology,Extending Database Technology* (Feb 2015).

[18] 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.

[19] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. 2013. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 214–223.

[20] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems*. Springer, 7–24.

[21] Benjamín Farías, Wim Martens, Carlos Rojas, and Domagoj Vrgoč. 2024. PathFinder: Returning Paths in Graph Queries. In *International Semantic Web Conference*. Springer, 135–154.

[22] Valeria Fionda, Giuseppe Pirrò, and Mariano P Consens. 2019. Querying knowledge graphs with extended property paths. *Semantic Web* 10, 6 (2019), 1127–1168.

[23] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.

[24] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[25] 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.

[26] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *Proceedings of the 2021 International Conference on Management of Data*. 645–657.

[27] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[28] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.

[29] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 681–697.

[30] Ansar Javed, Bibrak Qamar, Mohsan Jameel, Aamir Shafi, and Bryan Carpenter. 2016. Towards scalable java HPC with hybrid and native communication devices in MPJ express. *International Journal of Parallel Programming* 44, 6 (2016), 1142–1172.

[31] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux. In *Proceedings of the 2018 International Conference on Management of Data*. doi:10.1145/3183713.3196917

[32] Krys J Kochut and Maciej Janik. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. In *European Semantic Web Conference*. Springer, 145–159.

[33] Srdjan Komazec, Davide Cerri, and Dieter Fensel. 2012. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 58–68.

[34] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *Scientific and Statistical Database Management: 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings 24*. Springer, 177–194.

[35] Sofiane Lagraa, Martin Husák, Hamida Seba, Satyanarayana Vuppala, Radu State, and Moussa Ouedraogo. 2024. A review on graph-based approaches for network security monitoring and botnet detection. *International Journal of Information Security* 23, 1 (2024), 119–140.

[36] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*. Springer, 370–388.

[37] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph summarization methods and applications: A survey. *ACM computing surveys (CSUR)* 51, 3 (2018), 1–34.

[38] Wim Martens, Matthias Niewerth, Tina Popp, Stijn Vansummeren, and Domagoj Vrgoc. 2022. Representing paths in graph database pattern matching. *arXiv preprint arXiv:2207.13541* (2022).

[39] Kento Miura, Toshiyuki Amagasa, Hiroyuki Kitagawa, R Bordawekar, and T Lahiri. 2019. Accelerating Regular Path Queries using FPGA.. In *ADMS@ VLDB*. 47–54.

[40] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[41] 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.

[42] 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.

[43] Maurizio Nolé and Carlo Sartiani. 2016. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 1–12.

[44] Nigini Oliveira, Michael Muller, Nazareno Andrade, and Katharina Reinecke. 2018. The exchange in StackExchange: Divergences between Stack Overflow and its culturally diverse participants. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–22.

[45] 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.

[46] 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.

[47] 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.

[48] Thomas Pellissier Tanon, Gerhard Weikum, and F Yago Suchanek. [n. d.]. 4: A reason-able knowledge base. In *Proceedings of the European Semantic Web Conference*. 583–596.

[49] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 243–252.

[50] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[51] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2017. Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11, 4 (2017), 1–50.

[52] Frank Tetzel, Wolfgang Lehner, and Romans Kasperovics. 2020. Efficient Compilation of Regular Path Queries. *Datenbank-Spektrum* 20 (2020), 243–259.

[53] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[54] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[55] 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.

[56] Xin Wang, Junhu Wang, and Xiaowang Zhang. 2016. Efficient distributed regular path queries on rdf graphs using partial evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1933–1936.

[57] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 145–156.

[58] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2013. Evaluation of SPARQL Property Paths via Recursive SQL. *AMW* 1087 (2013).

[59] Siyuan Zhang, Zhenying He, Yinan Jing, Kai Zhang, and X Sean Wang. 2024. MWP: Multi-Window Parallel Evaluation of Regular Path Queries on Streaming Graphs. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[60] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. 2012. SRBench: a streaming RDF/SPARQL benchmark. In *The Semantic Web–ISWC 2012: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I 11*. Springer, 641–657.

[61] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1323–1334.

Received April 2025; revised July 2025; accepted August 2025