Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation

Anonymous

March 21, 2022

Abstract

Large workloads of event trend aggregation queries are widely deployed to derive high-level insights about current event trends in near real time. To speed-up the execution, we identify and leverage sharing opportunities from complex patterns with flat Kleene operators or even nested Kleene expressions. We propose GLORIA , a graph-based sharing optimizer for event trend aggregation. First, we map the sharing optimization problem to a graph path search problem in the GLORIA graph with execution costs encoded as weights. Second, we shrink the search space by applying cost-driven pruning principles that guarantee optimality of the reduced GLORIA graph in most cases. Lastly, we propose a path search algorithm that identifies the sharing plan with minimum execution costs. Our experimental study on three real-world data sets demonstrates that our GLORIA optimizer effectively reduces the search space, leading to 5-fold speed-up in optimization time. The optimized plan consistently reduces the query latency by 68%-93% compared to the plan generated by state-of-the-art approaches.

1 Introduction

Complex Event Processing has shown great promise in retrieving insights over high-velocity event streams in near real time for a broad range of domains ranging from transportation, finance, to public health. Applications in these domains often rely on complex nested event aggregation queries composed of Kleene [2, 7] and other event operators that express complex dependencies among event types to retrieve summarized information of interest. Unlike fixed length event sequences, the sequences matched by Kleene pattern queries, called *event trends* [27], can be arbitrarily long and expensive to compute [21, 29, 28]. It is thus becoming increasingly difficult to support high-performance event processing due to the rising number and complexity of event trend aggregation queries [3].

Multi-query optimization for event trend aggregation is a promising approach [21, 31, 28] that aims to reduce the query execution costs by sharing computations among queries in the workload. Given an event trend aggregation workload, a multi-query optimization technique must (1) identify sharing opportunities from a variety of complex query patterns (e.g., Kleene and nesting) and (2) decide how to leverage these sharing opportunities to truly benefit the execution of the given workload.

Motivation Example. Figure 1 shows a multi-query event trend aggregation workload in the food delivery scenario. An event type corresponds to an action made by the customer or the delivery driver, e.g. *AppOrder* or *WebOrder*, *Request* or *Travel*, etc. Each event in the stream is a tuple composed of a customer identifier, driver identifier, an action, a district, and a timestamp.

```
Query:
RETURN district, SUM(Travel.duration)
PATTERN P
WHERE [driver_id] GROUP_BY district
WINDOW 20min SLIDE 5 min
```

q_1: Request, (Pickup, Travel)+, Delivery
q_2: AppOrder, Request, (Pickup, Travel)+, Delivery
q_3: AppOrder, Request, (Pickup, Travel)+, Cancel

Figure 1: Event Trend Aggregation Workload

In Figure 1, we present the common clauses of the workload above, and show three different patterns for each query that each matches event trends of an order. In real-world, a delivery driver usually picks up several orders nearby before one delivery to shorten his/her travelling route, which allows arbitrary number of consecutive $(Pickup\ ,Travel)$ in an order event trend. Thus, query q_1 focuses on the orders that are placed on any end and finished by delivery. query q_2 focuses on the orders that are placed on the app end and finished by delivery, and q_3 tracks the order that are placed on the app end but cancelled by the customer. All three queries contain the sub-pattern $Request, (Pickup, Travel)+$, which requires sharing the Kleene sub-pattern. However, this is not the only sharing opportunity, notice that q_1, q_2 could share longer sub-pattern $(Request, (Pickup, Travel)+, Delivery)$, and q_2, q_3 have another longer sub-pattern SEQ sub-pattern $(AppOrder, Request, (Pickup, Travel)+)$. To share the queries with both Kleene and SEQ sub-patterns, together with multiple conflicting sharing opportunities is not a trivial task.

State-of-the-Art Approaches. Event trend aggregation, in general, is tackled by two execution strategies, two-step and online aggregation. Early works [17, 38, 27, 21] use a *two-step approach*, that first fully constructs and then aggregates event trends matched by a SEQ or Kleene pattern. While the event trend construction step may be shared by multiple pattern queries [17, 21], these two-step methods continue to suffer from an exponential complexity in the worst case due the explosive complexity of generating all event trends first [38, 27]. On the other hand, *online aggregation methods* [29, 31, 30, 34, 28] hold promise by pushing the later aggregation into the earlier pattern matching phase without first having to construct any event trends. This reduces the computation complexity from exponential to quadratic [29]. However, these state-of-the-art online aggregation methods suffer from two critical limitations as explained next.

First, Kleene patterns often can be very complex with nesting not only across SEQ but also Kleene patterns. Hence, most existing methods [21, 28, 30, 31] restrict or completely disallow Kleene patterns supported in the event trend aggregation. For example, MCEP [21, 30] only support the sharing of non-nested (flat) Kleene patterns with the pattern composed of only one single event type, while SHARON [31] supports sharing fixed-length SEQ patterns at best, i.e., no Kleene patterns. Such restrictions on the Kleene patterns greatly reduce the applicability of existing sharing methods on real-world query workloads [28].

Second, the state-of-the-art methods make strict assumptions to derive at some sharing decisions for multi-query event trend aggregation. For example, SHARON [31] introduces the concept of a sharing conflict, which does not allow one sub-pattern to participate in multiple sharing query groups. Similar to MCEP [21], HAMLET[28] only considers sharing opportunities among a special case Kleene sub-pattern, namely, one that is flat and only contains a single event type. These assumptions often result in sub-optimal sharing plans for event trend aggregation queries, since many sharing opportunities are unnecessarily missed. Table 1 summarizes exemplar approaches categorized by the three dimensions discussed above.

Approach	**Aggregation strategy**	**Kleene pattern type**	**Sharing decision**
MCEP [21]	two-step	restricted	flexible
SHARON [31]	online	–	restricted
GRETA [29]		general	–
HAMLET [28]		restricted	restricted
GLORIA (our)		**general**	**flexible**

Table 1: Event trend aggregation approaches.

Challenges. To tackle the problem of multi-event trend aggregation, we need to address the following challenges.

Query complexity. In real-world applications, a query workload often consists of Kleene queries with a sequence of event types and even with fully nested Kleene sub-patterns. A sharing optimizer needs to discover sharing opportunities among those complex Kleene queries, despite the exponential complexity of the problem [38, 27].

Sharing complexity. Sharing opportunities need to be fully exploited without rigid constraints. Moreover, a model is needed to accurately capture the computational overhead incurred by shared query executions. A sharing optimizer can only harvest the maximal sharing benefit when it is able to identify truly beneficial sharing opportunities among general event trend aggregation queries with Kleene and nesting.

Search complexity. Allowing flexible sharing among arbitrary Kleene queries increases the search space of possible sharing plans, making it prohibitively expensive to find the optimal one. To address this challenge, we need effective pruning strategies to reduce the search space without compromising the optimality.

Proposed Solution: GLORIA . Given a workload of event trend aggregation queries composed of nested SEQ and Kleene sub-patterns, the GLORIA optimizer generates a fine-grained sharing plan that decides which set of queries should share which sub-patterns depending on the event stream characteristics. Specifically, by mapping the sharing optimization problem into a graph path searching problem, the GLORIA optimizer generates a GLORIA graph that captures the sharing plan search space. Thanks to the mapping, the optimal path properties can be leveraged to prune the graph early on during its construction.

For SEQ pattern queries, we design a set of effective pruning rules applicable during the GLORIA graph construction itself to minimize the size of GLORIA graph. Moreover, these pruning rules can be further leveraged by the graph path searching algorithm to purge the number of candidate paths from the GLORIA graph. This allows our algorithm to generate the optimal workload sharing plan for SEQ pattern queries in linear time. For Kleene pattern queries, we exploit the proposed pruning rules with additional Kleene-specific rules to construct a compact GLORIA sub-graph for Kleene sub-patterns. The resulting compressed sub-graph is then connected to the existing partial GLORIA graph to generate the complete one.

Contributions. Our main contributions are as follows.

- We present a novel approach for optimizing the sharing plan for a workload of event trend aggregation queries with complex Kleene patterns. To our best knowledge, GLORIA is the first sharing optimizer that offers flexible sharing decisions on complex Kleene patterns using effective online aggregation.
- We introduce GLORIA graph to model a variety of sharing opportunities in a diverse multi-event trend aggregation workload. A cost model and a set of effective pruning rules are introduced to reduce the size of GLORIA graph during its construction. This allows us to transform the workload sharing problem into a path search problem.
- We design a path search algorithm to find a multi-event trend aggregation sharing plan from the GLORIA graph. For a given complex query workload, our algorithm achieves a linear-time complexity in the number of nodes and edges in the GLORIA graph, while still delivering optimality guarantees in most cases.
- The experimental evaluation on three public data sets demonstrates the effectiveness of our pruning principles and the quality of the produced workload sharing plan. Our sharing plan achieves significant performance improvement ranging from 3-fold to 1 order of magnitude over the state-of-the-art approaches.

Outline. The remainder of this paper is organized as follows. Section 2 describes the GLORIA query model and defines the multiple aggregation query sharing problem. Section 3 illustrates the GLORIA system framework first, then demonstrate the execution with our cost model. In Section 5, we introduce the core GLORIA graph model with pruning principles. In Section 6, we extend our GLORIA optimizer to support sharing optimization for general Kleene patterns. We present experiments, related work, and conclusion in Sections 8, 9, and 10, respectively.

2 Preliminaries

2.1 GLORIA Query Model

Definition 2.1 (Kleene Pattern). *A pattern P is in the form of E, P_1+, (NOT P_1), SEQ(P_1, P_2), $(P_1 \vee P_2)$, or $(P_1 \wedge P_2)$, where E is an event type, P_1, P_2 are patterns, $+$ is a Kleene plus, NOT is a negation, SEQ is an event sequence, $\vee$ a disjunction, and $\wedge$ a conjunction. P_1 and P_2 are called sub-patterns of P. If a pattern P contains a Kleene plus, P is called a Kleene pattern. If a Kleene operator is applied to the result of another Kleene pattern, then P is a nested Kleene pattern. Otherwise, P is a flat Kleene pattern.*

Definition 2.2 (Event Trend Aggregation Query). *An event trend aggregation query q consists of five clauses:*

- *RETURN clause: Aggregation result specification;*
- *PATTERN clause: Event sequence pattern P per Definition 2.1;*
- *WHERE clause: Predicates θ; (optional)*
- *GROUPBY clause: Grouping G; (optional)*
- *WITHIN/SLIDE clause: Window w.*

Notation	Description
Q	A workload of queries
$pe(E, q)$	Predecessor types of E w.r.t q
$tran(E_i, E_j)$	Transition btw. E_i and E_j in template
$expr(e_i, Q_j)$	Snapshot expression of e_i for Q_j
sp_{e_i}	Snapshot of an event e_i
$Pool(E_i, E_j)$	Pool of candidate transition sharing plans for transition (E_i, E_j)
n_k	Node n_k in $Pool(E_i, E_j)$ (i.e., a candidate transition sharing plan)
n^q_{st}	Start node of q in GLORIA graph
n^q_{ed}	End node of q in GLORIA graph
$edge(n_i, n_j)$	Edge between n_i and n_j
$n_i.d_s$	Minimum distance from all start nodes to n_i
$n_i.d_e$	Minimum distance from n_i to all end nodes
$P.w$	Weight of a path P in GLORIA graph

Table 2: Table of notations.

An event trend of query q is a result sequence of events tr = (e_1, . . ., e_k) that matches the pattern P. For any adjacent events e_i, e_{i+1} in tr, e_i is called the predecessor event of e_{i+1}. All events in a trend satisfy predicates θ, have the same values of grouping attributes G, and are within one window w. Table 2 summarizes notations.

Given query q and event trend tr of q, the event types $e_1.type$ and $e_k.type$ of the first and last events e_1 and e_k in tr must be starting and ending event types $start(q)$ and $end(q)$ in q, respectively.

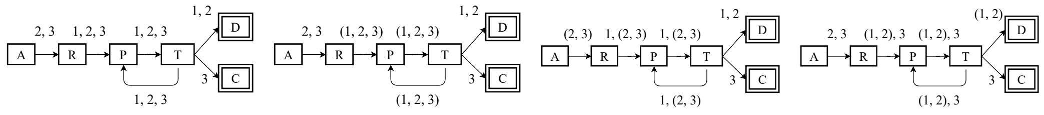
Definition 2.3 (Shareable Patterns)**.** *Given a pattern P and a workload Q of event trend aggregation queries, if P appears in the pattern clause of at least two queries, this pattern P is shareable.*

Example 2.4. *For queries with Kleene patterns q_1 = SEQ(A, B)+, q_2 = SEQ(C, SEQ(A, B)+)+, and q_3= SEQ((C, SEQ(A, B)+)+, E), the flat Kleene sub-pattern SEQ(A, B)+ is shareable by all three queries and nested Kleene sub-pattern SEQ(C, SEQ(A, B)+)+ by q_2 and q_3.*

Aggregation Functions. We focus on aggregation functions that can be computed incrementally[15]. COUNT(*) returns the number of matched event trends, MIN($E.attr$) and MAX($E.attr$) return the minimum or maximum of an attribute $attr$ of events of type E in all event trends matched by q, while SUM($E.attr$) and AVG($E.attr$) compute the sum or average of $attr$ of events of type E in all event trends matched by q. For simplicity, we use COUNT(*) as the default aggregation function in this paper. We discuss sharing other aggregation functions in Section **??**.

2.2 Multi-Event Trend Aggregation Sharing Problem

To facilitate the analysis of sharing opportunities, we represent a query workload as a Finite-State-Automaton [11, 13, 37, 38], called GLORIA *Template*. Each node in the template represents an event type in q. A transition from event type E_i to E_j represents a SEQ or Kleene operator between E_i and E_j, denoted as $tran(E_i, E_j)$. Event types that precede E_j in query q (i.e., there is a transition from E_i to E_j) are denoted as $pe(E_j, q)$. We adopt the state-of-the-art algorithm for this template construction [29].



(a) Template of Q (b) Sharing plan 1 (c) Sharing plan 2 (d) Sharing plan 3

Figure 2: GLORIA template and sharing plans of $Q = \{q_1, q_2, q_3\}$

Example 2.5. *Figure 2(a) shows the template of workload Q in Figure 1, where q_1 = SEQ(R, SEQ(P, T)+, D), q_2 = SEQ(A, R, SEQ(P, T)+, D), q_3 = SEQ(A, R, SEQ(P, T)+, C). This template reveals multiple sharing opportunities. Figures 2(b)-2(d) show three sharing plans.*

With respect to transitions, sharing plan 1 in Figure 2(b) shares q_1, q_2, q_3 on transitions $tran(R, P)$, $tran(P,T)$ and $tran(T,P)$, denoted as $(1,2,3)$ on the three transitions. Alternatively, plan 2 in Figure 2(c) shares q_2, q_3 on transitions $tran(R,P)$, $tran(P,T)$ and $tran(T,P)$, . Plan 3 in Figure 2(d) shares q_1, q_2 on transitions $tran(R,P)$, $tran(P,T)$, $tran(T,P)$ and $tran(T,D)$. Intuitively, a transitions[1] in the template can be shared by different queries.

This example raises three questions. (1) how to share the Kleene sub-patterns. (2) if consecutive transitions could have different shared queries. For example, $tran(A,R)$ shares $(1,2)$ as in Figure 2(c) but $tran(R,P)$ shares $(1,2,3)$ as in Figure 2(d). (3) assume we know the answers of the above questions, how to determine the cost and find the optimal workload sharing plan.

To obtain a sharing plan for a workload Q, we must associate a set of queries for sharing with each transition in the template. The set of queries shared together is called a *Q-set*.

Definition 2.6 (Transition Sharing Plan)**.** *A transition sharing plan partitions all queries associated with a transition into a set of Q-sets such that the queries in each Q-set share the execution of modeled by this transition.*

Definition 2.7 (Workload Sharing Plan)**.** *Given a* GLORIA *template of a workload Q, a workload sharing plan of Q consists of a set of transition sharing plans for all transitions in the template.*

Problem Statement. Given workload Q and a stream, the event trend aggregation sharing problem is to find a workload sharing plan composed of pattern sharing that executes Q with minimized average query latency[2]. The latency of a query $q \in Q$ is measured as the difference between the time point of the aggregation result output by the query q and the arrival time of the last event that contributed to this result.

Search Space. Given a transition (E_i, E_j) associated with m queries, determning *Q-sets* of these queries corresponds to partitioning the set of m elements into n ($n \leq m$) non-overlapping subsets that together cover the set m. The number of partitions is known as Bell-Number of m [19]:

$$B_m = \sum_{n=1}^{m} \left\{ \begin{matrix} m \\ n \end{matrix} \right\} = O(e^{e^m}) \tag{1}$$

Given a workload Q and its GLORIA template with k transitions, the number of transition sharing plans for each transition is $O(B_{|Q|})$ in the worst case (Equation 1). Since a workload sharing plan is a combination of transition sharing plans of all k transitions, the size of the search space S_Π of the workload sharing plan optimization problem Π is $O(B^k_{|Q|})$. In other words, enumerating all possibilities for a transition has an exponential time complexity in the worst case, i.e., it is too prohibitively expensive. Hence, an efficient and effective optimizer is needed to generate the optimized workload sharing plan.

3 GLORIA System

3.1 GLORIA Overview

Figure 3 depicts the GLORIA framework. The GLORIA optimizer takes as input a workload of queries and stream statistics. The query workload is represented by a template (Section 2.2). The optimizer transforms the query template into a GLORIA graph to compactly encode all possible sharing opportunities (Sections 4 and 5.1). This way, an optimal workload sharing plan corresponds to a path with the minimum weight in the GLORIA graph. To reduce the size of this search space, a set of cost-based pruning rules are utilized (Sections 5.2 and 6). The sharing plan finder applies a path search algorithm supported by additional pruning principles to find the final workload sharing plan (Sections 5.3 and 6). This plan finder features an efficient search time linear in the size of the graph.

The GLORIA executor encodes the optimal workload sharing plan produced by the GLORIA optimizer. The executor then incrementally computes trend aggregates by propagating intermediate aggregates from previously matched events to new events. These intermediate aggregates are captured as *snapshots* in the snapshot manager for sharing.

[1]The terms sub-pattern and transition are used interchangeably in this paper.

[2]Average latency is the total latency of the entire workload divided by the number of queries in the workload.

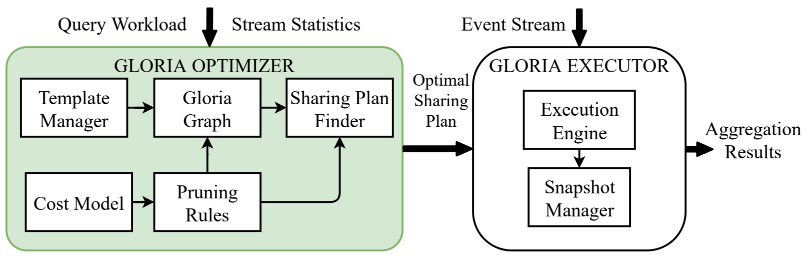


Figure 3: GLORIA framework.

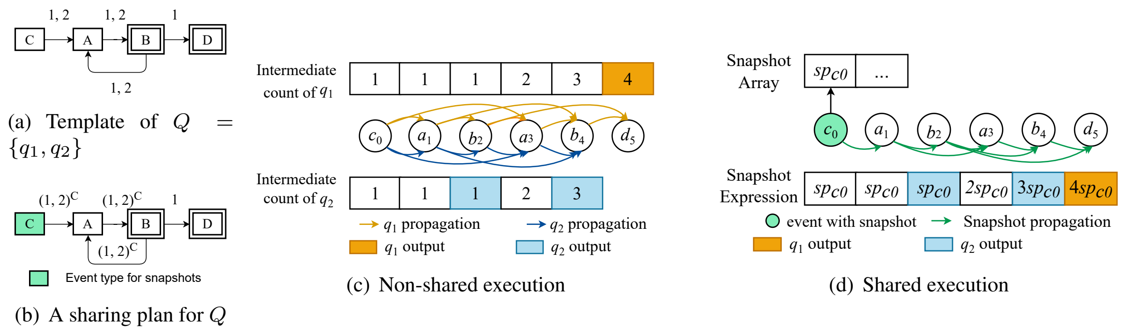


Figure 4: Non-shared and shared executions over an event stream $I = \{c_0, a_1, b_2, a_3, b_4, d_5\}$.

3.2 GLORIA Executor

In this paper, GLORIA executor adopts the state-of-art online aggregation methods [29, 28] to support both non-shared and shared executions. Figure 4 features an example template with both non-shared and shared executions. We use COUNT(∗) as the aggregation function as an example.

Non-shared Online Aggregation. Every event e maintains an intermediate aggregate for each query q in non-shared online aggregation, indicating the count of event trends matched by q and ending with e. During the execution, the count of e is incremented by the sum of the intermediate aggregates of the predecessor events of e that were matched by q (denoted $pe(e, q)$). If $e.type = end(q)$, this aggregate is output as the final result of q.

Example 3.1. *Figure 4(c) shows the non-shared execution of Q in Figure 4(a) over an event stream I. The arrows indicate the intermediate aggregate propagation for each event per query. When c_0 arrives, it starts a new event trend for both q_1 and q_2, and its intermediate aggregates are 1 for q_1 and q_2. When a_1 arrives, its intermediate aggregates are obtained by propagation from its predecessors c_0 for q_1 and q_2. For each following event, the intermediate aggregates are obtained by summing the values of the predecessors following the propagation for each query. Finally, when d_5 arrives, its intermediate aggregate is output as the final count $fcount = 4$ for q_1.*

The execution cost of q_1 and q_2 lies in the propagation of intermediate aggregates from the predecessors to the newly arrived matched events. For example, $cost(a_1, q_1)$ = $cost(a_1, q_2)$ = 1 and $cost(a_3, q_1)$ = $cost(a_3, q_2)$ = 2. As shown in Figure 4(c), the non-shared execution cost of all events of type A for Q (q_1 and q_2) is 6 (two arrows pointing to a_1 and four arrows pointing to a_2).

In general, the non-shared online aggregation execution cost of all events of type E for a given workload Q is

$$Cost_{nonshared}(E, Q) = \sum_{q \in Q} \sum_{E_p \in pe(E,q)} |E| \times |E_p| \tag{2}$$

where $|E|$ and $|E_p|$ denotes the number of E events and the one of each predecessor event of E for q, respectively.

Shared Online Aggregation. Non-shared online aggregation incurs re-computation for the common sub-pattern of multiple queries. As shown in Figure 4(c), the intermediate aggregate of c_0 and a_1 are propagated to the events a_1 to b_4 twice for q_1 and q_2. To avoid such re-computation, we exploit *Snapshots* [28] to support efficient sharing. A snapshot can be either a variable corresponding to an intermediate aggregate of a query or an expression composed of several snapshots. The GLORIA executor creates a snapshot for each event by summing the snapshots of its predecessors. Intuitively, a snapshot of e_i captures the number of event trends that can be extended by a new event e_j. Only one snapshot propagation from e_i to e_j is required for all queries sharing the common sub-pattern, instead of propagating all intermediate values of e_i. During the shared execution, a snapshot of an event e is only evaluated when $e.type = end(q)$ for $q \in Q$.

Example 3.2. *Figure 4(b) shows a workload sharing plan that shares q_1, q_2, using the snapshots of event type C, denoted as $(1,2)^C$, for three transitions $tran(C,A)$, $tran(A,B)$ and $tran(B,A)$. Figure 4(d) illustrates the corresponding shared execution. When c_0 arrives, it increments 1 for both q_1 and q_2. Then these values are stored into a new snapshot sp_{c_0}, which is inserted into a snapshot hash map. The snapshot expression of c_0 is set to sp_{c_0}. When a_1 and b_2 arrive, their predecessors c_0 and a_1 propagate their snapshot to a_1 and b_2, respectively. Since $b_2.type = end(q_2)$, its expression is evaluated for q_2 and returns 1 as the result. When a_3 arrives, instead of having c_0 to propagate its snapshot, it obtains c_0's information from its predecessors a_1 and b_2. Analogously, when d_5 arrives, b_2 and b_4 propagate their snapshots to d_5 and d_5's expression is evaluated to produce the final value 4 for q_1.*

The shared execution cost lies in the snapshot propagation and expression evaluation. The costs of value insertion and snapshot maintaining are respectively minimal. The former are constant in the number of queries, meanwhile the latter is O(1) for operations on a snapshot hash map. Since each event carries only one snapshot expression, the number of snapshot propagations for all events of type A is reduced from 6 to 3 as indicated in Figure 4(d). And three end events b_2, b_4 and d_5 require evaluations of the associated snapshots 3 times. This sharing plan saves 3 propagations compared to the non-shared execution. However, it requires 3 additional evaluations of the snapshots. Hence the sharing plan can only be beneficial when the saving from the snapshot propagation outweigh the snapshot evaluation overhead. Intuitively, a sharing benefit would be substantial when more queries and events are shared.

In general, the shared propagation cost of all events of type E for a given workload Q is

$$Cost_{share}(E,Q) = \sum_{E_p \in pe(E,q)} |E_p| \times expr(E_p, Q).len \tag{3}$$

where E_p denotes the event type of predecessors of the event type E, and $expr(E,q).len$ is the length of the snapshot expression of E_p for Q, which corresponds to the frequency of the event type of the snapshot. In case that E is an end event type of q, triggering the evaluation of the snapshot expression, the cost is:

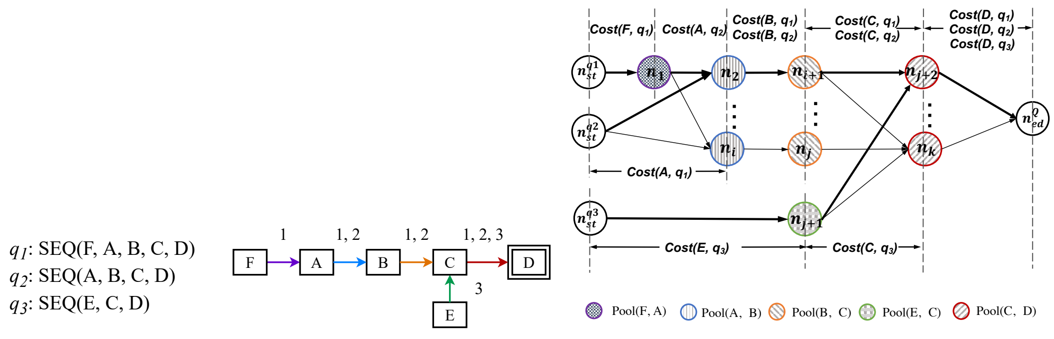
$$Cost_{eval}(E,Q) = \sum_{q \in Q} |E| \times expr(E,q).len \tag{4}$$

4 Gloria Graph Model

We now introduce our Gloria graph model to transform the workload sharing plan problem into an optimal path problem. Given a template, a workload sharing plan decides which queries are shared on each transition in the template. We use the Gloria graph to capture the search space of all workload sharing plans.

Definition 4.1 (Gloria Graph)**.** *A Gloria graph is a weighted directed graph with a set of nodes and a set of directed edges. A node n_k represents either a transition sharing plan of $tran(E_i, E_j)$ in which E_i and E_j are two adjacent event types, or a start node n^q_{st} to indicate the start of query q or an end node n^q_{ed} to indicate the end of q. A pool $Pool(E_i, E_j)$ consists of all nodes, i.e., candidate sharing plans, for $tran(E_i, E_j)$. A directed weighted edge $edge(n_k, n_m)$ connects node n_k to n_m, if they belong to two consecutive pools. Let $\bar{Q}$ be the common queries in n_k ($tran(E_i, E_j)$) and in n_m ($tran(E_j, E_h)$), then the weight of the edge $edge(n_k, n_m)$ represents the execution costs ($Cost(E_j, \bar{Q})$) of E_j for $\bar{Q}$, when applying the transition sharing plans n_k and n_m.*

Example 4.2. *Figure 5 depicts the template of a workload $Q = \{q_1, q_2, q_3\}$ where $q_1 =$ SEQ(F,A,B,C,D), $q_2 =$ SEQ(A,B,C,D) and $q_3 =$ SEQ(E,C,D) and its corresponding Gloria graph. For example, n_1 is the only sharing plan in $Pool(F,A)$ for $tran(F,A)$. The nodes (n_2, ..., n_i) are the sharing plans in $Pool(A,B)$ for $tran(A,B)$. To indicate the start and end of each query, we add three start nodes (i.e., $n^{q_1}_{st}$, $n^{q_2}_{st}$, and $n^{q_3}_{st}$) and a common end node n^Q_{ed}, since q_1 to q_3 start with different event types but end with the same one. For pools of consecutive transitions like $Pool(A,B)$ and $Pool(B,C)$, their nodes are connected such that each $n_k \in Pool(A,B)$ has an outgoing edge to each $n_m \in Pool(B,C)$.*



(a) Workload (b) Template (c) GLORIA graph

Figure 5: GLORIA graph model for $Q = \{q_1, q_2, q_3\}$.

Intuitively, a GLORIA path structure, explained below, corresponds to a selection of sharing decisions that completely covers the transitions of the workload.

Definition 4.3 (GLORIA Path)**.** *Given a* GLORIA *graph, a* GLORIA *path structure P (or, in short, path P) consists of a list of edges in the* GLORIA *graph, starting from all start nodes, connecting one node from each pool, and ending with all end nodes.*

The bold lines in Figure 5(c) illustrate an example path, namely one workload sharing plan for Q. It provides one sharing plan for each transition.

Lemma 4.4. *The weight $P.w$ of a* GLORIA *path P corresponds to the execution cost of the workload sharing plan that P represents.*

Proof. Given a path P, and an arbitrary node $n_k \in pool(E_j, E_h)$ on P, by the definition of the edge weight in Definition 4.1, the sum of weights of all incoming edges for n_k on P covers the execution cost of E_j for queries on $tran(E_j, E_h)$. Based on that, by the definition of path, P visits every pool in the graph that every node covers the execution cost of an event type for a transition. Therefore, given a path P, its weight $P.w$ exactly captures the execution cost of all event types for all transitions, which equals the execution cost of the workload sharing plan that P represents. □

Lemma 4.5. *The paths in the* GLORIA *graph cover all possible workload sharing plans.*

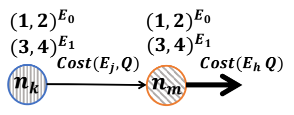
Proof. We prove this by contradiction. Assume that there is one workload sharing plan that is not a path in the GLORIA graph. Since each node represents a transition sharing plan, this workload sharing plan can be represented as a set of nodes that each node is from a unique pool. Also, since this workload is not a path, there must be at least two nodes from consecutive pools that are not connected in the GLORIA graph. However, according to the definition of GLORIA graph in Definition 4.1, every pair of nodes from consecutive pools is connected. Therefore, such workload sharing plan does not exit. □

Lemma 4.6. *Let $\bar{P}$ denote the path with the minimal path weight in the* GLORIA *graph. $\bar{P}$ corresponds to the optimal workload sharing plan.*

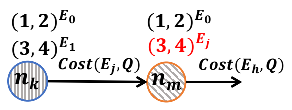
Proof. The proof can be inducted from Lemma 4.4 and 4.5. □

5 GLORIA Optimizer

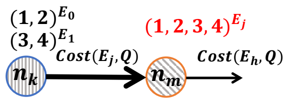
In this section, we introduce the GLORIA optimizer based on the GLORIA graph model. We propose three principles for search space reduction that limit the number of nodes (i.e., transition sharing plans) created in the graph. We also design two classes of pruning rules for nodes and edges respectively to further reduce the size of the GLORIA graph. With the pruned GLORIA graph, our path search algorithm finds the optimized workload sharing plan equal to the one in the unpruned GLORIA graph. We illustrate the core of the GLORIA optimizer on SEQ sub-patterns below and the extended optimizer also covering Kleenesub-patterns in Section 6.



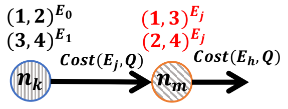
(a) Node Generation Principle (Reuse)



(b) Node Generation Principle (Local Snapshot)



(c) Node Generation Principle (Merge)



(d) Other Nodes

Figure 6: Node generation

5.1 Node Generation

According to our cost model in Equation 3, the sharing benefit lies in the single snapshot expression propagation for a *Q-set*. To maximize the sharing benefit, a *Q-set* should be maintained for as many transitions as possible. Otherwise, if two consecutive transitions have sharing plans that shares different *Q-set*s, the executor keeps applying the expensive evaluation to adapt to different sharing plans, which jeopardizes the sharing benefit. To maintain a stable sharing status for consecutive transitions, the optimizer generates a node $n_m \in Pool(E_j, E_h)$ from a node $n_k \in Pool(E_i, E_j)$.

A node $n_k \in Pool(E_i, E_j)$ can generate multiple nodes in $Pool(E_j, E_h)$. However, not all nodes are worth being generated. Recall that the goal of the optimizer is to find the optimal path in GLORIA graph with minimum weight, if such path passes n_k and one node $n_m \in Pool(E_j, E_h)$, and the weights of both the incoming and outgoing edge of n_m are guaranteed larger than other nodes in the same pool. Such local expensive n_m without potential saving opportunities doesn't have to be generated, compared with other nodes in $Pool(E_j, E_h)$.

Example 5.1. *Figure 6 shows four cases of $n_m \in Pool(E_j, E_h)$ that can be generated from the same n_k. The respective weight of the incoming and the outgoing of n_m is represented by the line thickness. Compared with other three cases, n_m in Figure 6(d) has heavy weights of both incoming and outgoing edges, which corresponds to $Cost(E_j, Q)$ and $Cost(E_h, Q)$ respectively. This local expensive n_m brings no potential sharing benefits.*

Lemma 5.2. *Given $n_k \in Pool(E_i, E_j)$ with Q-sets, when generating $n_m \in Pool(E_j, E_h)$, the sharing plans that (1) increase the number of Q-sets or (2) maintain the number but shuffle the Q-sets of n_k can be safely pruned.*

Proof. Let $n_k \in Pool(E_i, E_j)$ have a set of *Q-sets* $\{\bar{Q_1}, \bar{Q_2} \dots \bar{Q_x}\}$ that each *Q-set* has an event type as the snapshots $\{E_1, E_2 \dots E_x\}$. We prove that for case 1 and case 2, we can always find another node $n'_m \in Pool(E_j, E_h)$ that can replace n_m, where the path passing n'_m has a lighter weight that the path passing n_m.

1) Case 1

Suppose when generating n_m from n_k, from 1 to x-1, every *Q-set* $\bar{Q_i}$ with its event type of snapshots E_i is kept, but $\bar{Q_x}$ is split into two *Q-sets* $\bar{Q_x^1}$ and $\bar{Q_x^2}$ with event type of snapshots E_x^1 and E_x^2 respectively, where $E_x^1, E_x^2 \in \{E_j, E_x\}$. Therefore, n_m have the sharing plan of $\{\bar{Q_1}, \bar{Q_2} \dots \bar{Q_{x-1}}, \bar{Q_x^1}, \bar{Q_x^2}\}$ with respective event type as snapshots $\{E_1, E_2 \dots E_{x-1}, E_x^1, E_x^2\}$.

Let $n'_m \in Pool(E_j, E_h)$ be another generated node that keeps all *Q-set*s with respective event types of snapshots $\{E_1, E_2 \dots E'_x\}$ where $E'_x \in \{E_j, E_x\}$,

By using the sharing plan of n_k, the snapshot expression of E_j is computed for each *Q-set* $\bar{Q_1}$ to $\bar{Q_x}$. Then to split *Q-set* $\bar{Q_x}$ into $\bar{Q_x^1}$ and $\bar{Q_x^2}$, the snapshot expression of $\bar{Q_x}$ needs to be evaluated. Together, the weight of the incoming edge of n_m computes as follow:

$$\begin{aligned} edge(n_k, n_m).w &= Cost(E_j, Q) \\ &= \sum_{1 \le i \le x} Cost_{share}(E_j, \bar{Q_i}) + \sum_{E_x^i == E_j} Cost_{eval}(E_j, \bar{Q_x^i}) \end{aligned} \tag{5}$$

Similarly, for n'_m, the snapshot expression of E_j is computed by using the sharing plan of n_k, which makes the cost of sharing execution the same with the equation above. For the evaluation, if the new event type of snapshot E'_x does not equal to the old one E_x, the expressions of $\bar{Q_x}$ needs to be evaluated. Together, the weight of the incoming edge of n'_m computes as follow:

$$\begin{aligned} edge(n_k, n'_m).w &= Cost(E_j, Q) \\ &= \sum_{1 \le i \le x} Cost_{share}(E_j, \bar{Q_i}) + (E'_x == E_j) ? Cost_{eval}(E_j, \bar{Q_x}) : 0 \end{aligned} \tag{6}$$

Therefore, the difference between $edge(n_k, n'_m).w$ and $edge(n_k, n_m).w$ is:

$$edge(n_k, n'_m).w - edge(n_k, n_m).w = \begin{cases} Cost_{eval}(E_j, \bar{Q_x}) - \sum_{E^i_x == E_j} Cost_{eval}(E_j, \bar{Q^i_x}) & E'_x == E_j \\ -\sum_{E^i_x == E_j} Cost_{eval}(E_j, \bar{Q^i_x}) & E'_x ! = E_j \end{cases} \tag{7}$$

Since $\bar{Q^1_x} \cup \bar{Q^2_x} = \bar{Q_x}$, according to Equation 4, we have:

$$edge(n_k, n'_m).w - edge(n_k, n_m).w \le 0 \tag{8}$$

Then we prove the weight of outgoing edge of n'_m is smaller than the weight of outgoing edge of n_m. Let n_l be an arbitrary node that n'_m and n_m connect to. We prove that for any n_l, $edge(n'_m, n_l).w - edge(n_m, n_l).w \le 0$.

Since the exact sharing plan of n_l is unknown, we denote the evaluation cost on $edge(n_m, n_l)$ as $Cost_{eval}$ and the evaluation cost on $edge(n'_m, n_l)$ as $Cost'_{eval}$.

The weight of $edge(n_m, n_l)$ computes as follow:

$$\begin{aligned} edge(n_m, n_l).w &= Cost(E_h, Q) \\ &= \sum_{1 \le i \le x-1} Cost_{share}(E_h, \bar{Q_i}) + Cost_{share}(E_h, \bar{Q^1_x}) + Cost_{share}(E_h, \bar{Q^2_x}) + Cost_{eval} \end{aligned} \tag{9}$$

The weight of $edge(n'_m, n_l)$ computes as follow:

$$\begin{aligned} edge(n'_m, n_l).w &= Cost(E_h, Q) \\ &= \sum_{1 \le i \le x-1} Cost_{share}(E_h, \bar{Q_i}) + Cost_{share}(E_h, \bar{Q_x}) + Cost'_{eval} \end{aligned} \tag{10}$$

Therefore, the difference between $edge(n'_m, n_l).w$ and $edge(n_m, n_l).w$ is:

$$\begin{aligned} edge(n'_m, n_l).w - edge(n_m, n_l).w &= Cost_{share}(E_h, \bar{Q_x}) - Cost_{share}(E_h, \bar{Q^1_x}) - Cost_{share}(E_h, \bar{Q^2_x})) \\ &\quad + (Cost_{eval} - Cost'_{eval}) \\ &= |E_j| \times |E'_x| - |E_j| \times |E^1_x| - |E_j| \times |E^2_x| + (Cost_{eval} - Cost'_{eval}) \\ &= |E_j| \times (|E'_x| - |E^1_x| - |E^2_x|) + (Cost_{eval} - Cost'_{eval}) \end{aligned} \tag{11}$$

Since $E'_x, E^1_x, E^2_x \in \{E_j, E_x\}$, when $E'_x = E^1_x = E^2_x$, the first item in Equation 11 is smaller than 0. This means, for $1 \le i \le x - 1$, n_m and n'_m have the same *Q-sets* $\bar{Q_i}$ with the same event type of snapshots E_i. Also, the last two *Q-sets* $\bar{Q^1_x}$ and $\bar{Q^2_x}$ of n_m have the same event type of snapshots with the last *Q-set* $\bar{Q_x}$ of n'_m, where $\bar{Q_x} = \bar{Q^1_x} \cup \bar{Q^2_x}$. The identical event types of snapshots means that, for n_l, no matter which queries will be evaluated on $edge(n_m, n_l)$, the same queries will be evaluated on $edge(n_m, n_l)$ with the same evaluation cost. Therefore, the second item $(Cost_{eval} - Cost'_{eval})$ is equal to 0.

Together, we have:

$$edge(n'_m, n_l).w - edge(n_m, n_l).w < 0, E'_x = E^1_x = E^2_x \tag{12}$$

Combined with Equation 8, we prove that when $E'_x = E^1_x = E^2_x$, when connecting to the same nodes n_k and n_l, node n'_m has lighter weighted incoming and outgoing edges than n_m. Therefore, n_m can be safely pruned.

2) Case 2

Similarly with case 1, we prove that a newly generated node $n_m \in Pool(E_j, E_h)$ that keeps the number but shuffles the *Q-set*s of n_k has heavier weighted incoming and outgoing edges than a node n'_m which merges all *Q-set*s.

Suppose when generating n_m from n_k, all *Q-set*s are shuffled that n_m has a new set of *Q-set*s $\{\bar{Q'_1}, \bar{Q'_2} \ldots \bar{Q'_x}\}$. Since all *Q-set*s are newly generated, the event type of snapshots for each *Q-set* can only be E_j. Let n'_m be another node in $Pool(E_j, E_h)$ that merges all *Q-set*s of n_k, n'_m has one *Q-set* $\bar{Q}$ that $\bar{Q} = \bigcup \bar{Q'_i} = \bigcup \bar{Q}_i (1 \leq i \leq x)$. The merged *Q-set* $\bar{Q}$ has E_j as the event type of snapshots.

The weight of incoming edge of n_m computes as follow:

$$\begin{aligned} edge(n_k, n_m).w &= Cost(E_j, Q) \\ &= \sum_{1 \leq i \leq x} Cost_{share}(E_j, \bar{Q}_i) + \sum_{1 \leq i \leq x} Cost_{eval}(E_j, \bar{Q}_i) \end{aligned} \tag{13}$$

Since n_m and n'_m both needs to evaluate the snapshot expression of all queries into values, the weight of the incoming edge of n'_m is the same with n_m.

$$\begin{aligned} edge(n_k, n'_m).w &= Cost(E_j, Q) \\ &= \sum_{1 \leq i \leq x} Cost_{share}(E_j, \bar{Q}_i) + \sum_{1 \leq i \leq x} Cost_{eval}(E_j, \bar{Q}_i) \end{aligned} \tag{14}$$

Therefore, the difference between $edge(n_k, n'_m).w$ and $edge(n_k, n_m).w$ is:

$$edge(n_k, n'_m).w - edge(n_k, n_m).w = 0 \tag{15}$$

Then we prove the weight of outgoing edge of n'_m is smaller than the weight of outgoing edge of n_m. Let n_l be an arbitrary node that n'_m and n_m connect to. We prove that for any n_l, $edge(n'_m, n_l).w - edge(n_m, n_l).w \leq 0$. Since the exact sharing plan of n_l is unknown, we denote the evaluation cost on $edge(n_m, n_l)$ as $Cost_{eval}$ and the evaluation cost on $edge(n'_m, n_l)$ as $Cost'_{eval}$.

The weight of $edge(n_m, n_l)$ computes as follow:

$$\begin{aligned} edge(n_m, n_l).w &= Cost(E_h, Q) \\ &= \sum_{1 \leq i \leq x} Cost_{share}(E_h, \bar{Q'_i}) + Cost_{eval} \end{aligned} \tag{16}$$

The weight of $edge(n'_m, n_l)$ computes as follow:

$$\begin{aligned} edge(n'_m, n_l).w &= Cost(E_h, Q) \\ &= Cost_{share}(E_h, \bar{Q}) + Cost'_{eval} \end{aligned} \tag{17}$$

Therefore, the difference between $edge(n'_m, n_l).w$ and $edge(n_m, n_l).w$ is:

$$\begin{aligned} edge(n'_m, n_l).w - edge(n_m, n_l).w &= \sum_{1 \leq i \leq x} Cost_{share}(E_h, \bar{Q'_i}) - Cost_{share}(E_h, \bar{Q}) \\ &+ (Cost_{eval} - Cost'_{eval}) \\ &= (\sum_{1 \leq i \leq x} |E_j| \times 1 - |E_j| \times 1) + (Cost_{eval} - Cost'_{eval}) \end{aligned} \tag{18}$$

The first item in Equation 18 is always smaller than 0. For evaluation cost, similar with case 1, even though n'_m and n_m has different *Q-set*s, but they have the same event type of snapshots for all *Q-set*s. This means that no matter which queries will be evaluated on $edge(n_m, n_l)$, the same queries will be evaluated on $edge(n'_m, n_l)$ with the same evaluation cost. Therefore, the second item equals to 0, which makes Equation 18 smaller than 0.

Together with Equation 15, we prove that compared with n'_m, n_m can always be safely pruned.

□

Lemma 5.2 removes the local and global expensive nodes. Then we focus on the nodes that could be visited by the optimal path potentially. A node can be visited only if it has a light incoming edge or a light outgoing edge. Based on this observation, we propose three node generation principles.

Given a node $n_k \in Pool(E_i, E_j)$, a $n_m \in Pool(E_j, E_h)$ has the minimum incoming edge weight $edge(n_k, n_m).w$ when it reuses all ***Q-set***s and snapshots from n_k, which avoids the expensive evaluation.

Node Generation Principle 1 (Reuse). Given a node $n_k \in Pool(\ E_i, E_j)$, a node $n_m \in Pool(E_j, E_h)$ is generated so that n_m reuses ***Q-set***(s) and snapshots of ***Q-set***(s) from n_k.

Example 5.3. *Figure 6(a) shows an example of generating $n_m \in Pool(E_j, E_h)$ from $n_k \in Pool(E_i, E_j)$. n_k is sharing two Q-sets $\bar{Q}_1 = (1, 2)$ with snapshots E_0 and $\bar{Q}_2 = (3, 4)$ with snapshots of E_1. By the Node generation principle (Reuse), n_m reuses all the Q-sets together with the snapshots. According to our cost model in Equation 3 and the weight definition, the weight of edge is:*

$$\begin{aligned} edge(n_k, n_m).w &= Cost(E_j, Q) \\ &= Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \end{aligned} \tag{19}$$

This principle does not consider the weight of its outgoing edges. So a path passing this n_m could have a light weight before n_m but have a heavy weight after n_m.

Alternatively, instead of minimizing the weight of the incoming edge, a node n_m with heavier incoming edge may have a lighter outgoing edge, which corresponds to lower execution cost of E_h. According to Equation 3, the saving from sharing comes from two aspects. Either sharing the same ***Q-set***s with fewer snapshots which shorten the expression length, or sharing with fewer but larger ***Q-set***s. Therefore, when the optimizer generates a node n_m in the GLORIA graph, the expensive evaluation for E_j could be allowed, which increases the weight of incoming edge, when n_m brings the above two saving opportunities for E_h.

Therefore, $n_m \in Pool(E_j, E_h)$ can choose to share the same ***Q-set*** with n_k but create local snapshots of E_j, if E_j has lower frequency than the event type of snapshots used in n_k. Such snapshot replacing shortens the length of snapshot expressions of E_j and E_h, which reduces the execution cost of E_h.

Node Generation Principle 2 (Local Snapshots). Given a node $n_k \in Pool(E_i, E_j)$, a node $n_m \in Pool(E_j, E_h)$ is generated that it reuses the ***Q-set***(s) of n_k but creates local snapshots of E_j, if E_j has a lower frequency than the event type of old snapshots.

Example 5.4. *Figure 6(b) shows an example of generating n_m from n_k with local snapshots of E_j for Q-set $(3, 4)$. During execution for this Q-set, each event e of E_j sums the snapshot expressions of predecessors, which is an expression of snapshots of E_1, then this expression is evaluated to values and stored into a new local snapshot sp_e. This evaluation increases the weight of edge $edge(n_k, n_m)$ which is:*

$$\begin{aligned} edge(n_k, n_m).w = &Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \\ &+ Cost_{eval}(E_j, \bar{Q}_2) \end{aligned} \tag{20}$$

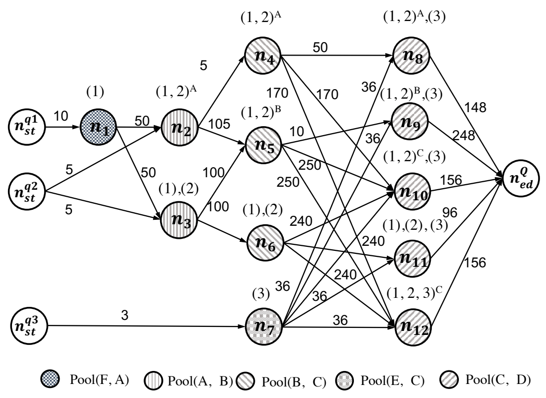
Compared with Equation 19, n_m in Figure 6(b) has a heavier incoming edge than Figure 6(a) but a lighter outgoing edge.

Another saving opportunity comes from merging ***Q-set***s. In this case, n_k merges all ***Q-set***s of n_k with local snapshots E_j. Such merging brings evaluation cost of E_j for every ***Q-set***, but could reduce the execution cost of E_h since the number of propagations for multiple ***Q-set***s is reduced to 1.

Node Generation Principle 3 (Merging). Given a node $n_k \in Pool(E_i\ , E_j)$, a node $n_m \in Pool(E_j, E_h)$ is generated that it merges the ***Q-set***s of n_k with local snapshots of E_j.

Example 5.5. *Figure 6(c) shows an example of generating n_m from n_k by merging Q-sets. According to our cost model, the weight of $edge(n_k, n_m)$ is:*

$$\begin{aligned} edge(n_k, n_m).w = &Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \\ &+ Cost_{eval}(E_j, \bar{Q}_1) + Cost_{eval}(E_j, \bar{Q}_2) \end{aligned} \tag{21}$$


(a) Full Gloria Graph

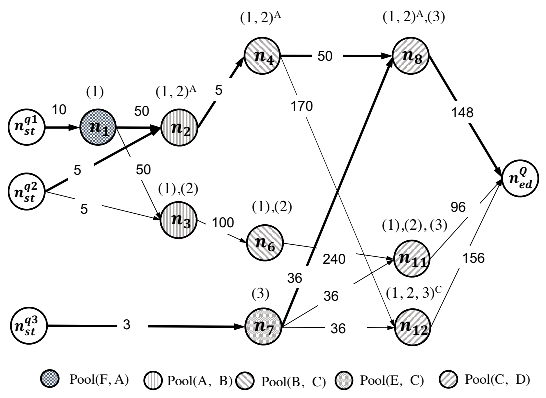

(b) Pruned Gloria Graph With Optimal Path

Figure 7: GLORIA Graph with and without pruning
Stream statistics: $|F| = 10$, $|A| = 5$, $|B| = 10$, $|C| = 12$, $|D| = 4$, $|E| = 3$

Compared with Equation 19 and 20, the $edge(n_k, n_m)$ has the heaviest weight for merging, but n_m may have the minimum weight of outgoing edge compared with the other two cases.

Node Generation From Multiple Proceeding Pools. If there are multiple pools $Pool(E_i, E_j)$ that $E_i \in pe(E_j, Q)$, each $n_k \in Pool(E_i, E_j)$ can generate part of n_m by above principles. Every combination of these parts forms a n_m. The optimizer applies node generation rules on *Q-set*s so the number of preceding pools doesn't affect the node generation.

GLORIA Graph Construction. The GLORIA graph is constructed from start nodes, pool by pool, to all end nodes. For each $n_k \in Pool(E_i, E_j)$, the optimizer generates multiple $n_m \in Pool(E_j, E_h)$, following one of node generation principles, together with the edges and the corresponding weights. To compare the sharing with non-sharing, we always generate a n_m of non-sharing from n_k that also non-shares.

Example 5.6. *Figure 7(a) shows the* GLORIA *graph of template in Figure 5(b). Each edge is labelled with its weight. Starting from start node n_{st}^{q1} for q_1, $Pool(F, A)$ has only candidate $n_1 = (1)$. In $Pool(A, B)$, n_2 is generated by merging q_1, q_2 following Node generation principle (Merging) and n_3 is a non-sharing plan. In $Pool(B, C)$, n_4 can be generated from n_2 by Node generation principle (Reuse). n_5 can be generated from n_2 by Node generation principle (Local Snapshot) or from n_3 by Node generation principle (Merging).*

Analogously, $Pool(C, D)$ can be generated based on $Pool(B, C)$ and $Pool(E, C)$, following the node generation rules. Since D is the ending event type for q_1 to q_3, all nodes n_8 to n_{12} in $Pool(C, D)$ are connected to a common end node st_{ed}^{Q}.

5.2 Progressive GLORIA Graph Pruning

Instead of pruning the GLORIA graph in a post-processing step, each pool is pruned immediately during the graph construction. Such progressive pruning process reduces the number of nodes in each pool, preventing the graph from exploding in the early stage as well as benefit the final path searching. Figure 7(b) shows the actual GLORIA graph we generate without losing optimality.

If a node has multiple ways to be generated from the start nodes and the optimal path visits this node, only the way with the minimum weight could be the optimal path. Consider n_5 in Figure 7(a) as a example, it could be generated from n_2 by *Node generation principle (Reuse)*, or from n_3 by *Node generation principle (Local Snapshot)*. Each way is represented as a path from $n_{st}^{q_1}$ and $n_{st}^{q_2}$ to n_5. By comparing the weights of the paths, we can find the optimal way to generate n_5, so that other paths to n_5 can be safely pruned. Both the path and its weight is stored in the node as the distance from start nodes to itself, denoted as $n_m.d_s$. Let $n_m \in Pool(E_j, E_h)$ be a node, Q be the queries on $tran(E_j, E_h)$, the distance $n_m.d_s$ can be computed as follows:

$$n_m.d_s = \sum_{E_i} min\{n_k.d_s + edge(n_k, n_m).w\}, \quad n_k \in Pool(E_i, E_j), E_i \in pe(E_j, Q) \tag{22}$$

Edge Pruning Principle. Given a node $n_m \in Pool(E_j, E_h)$, for incoming edges of n_m that comes from the same pool $Pool(E_i, E_j)$, as in Equation 22, only the edge which provides the minimum distance of n_m to all start nodes is kept. Other edges to n_m can be safely pruned.

Example 5.7. *Consider $Pool(B, C)$ in Figure 7(a) as the targeting pool. Table 3 lists the distances to all star nodes for all nodes. Nodes n_1 to n_3 do not need to be edge pruned. For n_5, there are two edges from $Pool(A, B)$, $edge(n_2, n_5)$ and $edge(n_3, n_5)$. By Equation 22, $n_2.d_s + edge(n_2, n_5).w = 170$ and $n_3.d_s + edge(n_3, n_5).w = 165$. Therefore, $edge(n_2, n_5)$ is pruned and $n_5.d_s$ is set to 165.*

Lemma 5.8. *The edge pruning principle does not discard any optimal path in the* GLORIA *graph.*

Proof. We prove this lemma by contradiction. Given a node $n \in Pool(E_j, E_h)$, let n_0 and n_1 be two nodes from $Pool(E_i, E_j)$ with edges $edge(n_0, n)$ and $edge(n_1, n)$ respectively. Assume $n_0.d_s + edge(n_0, n).w < n_1.d_s + edge(n_1, n).w$. If the optimal path $\bar{P}$ passes n_1 and n along $edge(n_1, n)$, then $\bar{P}$ is consisted of two parts P_1 and P_2 where P_1 is from all start nodes to n and P_2 is from n to all end nodes. $\bar{P}$'s weight can be computed as follows:

$$\begin{aligned} \bar{P}.w &= P_1.w + P_2.w \\ &= n_0.d_s + edge(n_0, n).w + P_2.w \end{aligned} \tag{23}$$

Instead, by visiting n through n_0, $\hat{P}.w = n_1.d_s + edge(n_1, n).w + P_2.w$ and $\hat{P}.w < \bar{P}.w$. Therefore, $\bar{P}$ cannot be the optimal path. □

Pool	**Node**	d_s	**Pruned**
$Pool(F, A)$	n_1	10	Keep
$Pool(A, B)$	n_2	65	Keep
	n_3	65	Keep
$Pool(B, C)$	n_4	70	Keep
	n_5	165	**Pruned**
	n_6	165	Keep
$Pool(E, C)$	n_7	3	Keep
$Pool(C, D)$	n_8	159	Keep
	n_9	-	-
	n_{10}	279	**Pruned**
	n_{11}	444	Keep
	n_{12}	279	Keep

Table 3: d_s of nodes in GLORIA graph

After edge pruning, every node obtain a unique d_s, then we consider node pruning. Given two nodes n_k and n_m in the same pool, if n_m is known to have larger distance to all start nodes d_s, as well as a larger distance to all end nodes d_e, the path passing n_m has heavier weight than the path passing n_k, so n_m can be pruned immediately.

Definition 5.9 (*Comparable Nodes*)**.** *Given two nodes n_k and n_m in the same pool, n_k and n_m are comparable, if they have the same Q-sets.*

Given two comparable nodes, we can estimate relative magnitude of d_e of two nodes by the number of snapshots that each *Q-set* is carrying.

Node Pruning Principle. Given two comparable nodes n_k, n_m in the same pool. If for every *Q-set*, n_k uses snapshots of less frequent event type than n_m, then
n_k has a smaller distance to start nodes than n_m. The node n_m can be safely pruned compared with n_k, if n_k has smaller distances to both start nodes and end nodes, denoted as $n_k.d_e < n_m.d_e$ and $n_k.d_s < n_m.d_s$.

Example 5.10. *Consider n_4 and n_5 in Figure 7(a) as an example. They both have the same Q-set $(1, 2)$ but with snapshots of different event types A and B respectively, where $|A| < |B|$. According to Node pruning principle, n_4 has a smaller distance to end nodes than n_5. Also, according to Table 3, n_4 also has a smaller distance to start nodes than n_5. Thus, n_5 can be pruned compared with n_4, denoted as $n_5(p)$ in the table.*

Lemma 5.11. *The node pruning principle does not exclude any optimal path on* GLORIA *graph.*

Lemma 5.11 can be proven by applying the cost model. Given two nodes n_k, n_m in $Pool(E_j, E_h)$ that shares $\bar{Q}$ with snapshots of event types A and B respectively, where $|A| < |B|$. According to Equation 3, due to shorter snapshot expressions, the sharing cost of E_h applying n_k is smaller than the cost of E_h applying n_m. If evaluated, according to Equation 4, the evaluation cost with snapshots of A is smaller than evaluation cost with snapshots of B. Thus, the path from n_k to all end nodes is guaranteed to be lighter than the path from n_m. Also, since $n_k.d_s < n_m.d_s$, the path passing n_k is guaranteed to be lighter than the path passing n_m. n_m can be safely pruned.

After n_5 is pruned, all its incoming edges are pruned in consequence. Then during the construction of $Pool(C, D)$, n_9 will not be constructed since n_5 is its only source of edge from $Pool(B, C)$. Analogously, during the pruning of $Pool(C, D)$, we first apply edge pruning for all nodes, then after node pruning, n_{10} is pruned compared with n_8. Table 3 shows the pruning status of each node.

Figure 7(b) shows the pruned graph after the last $Pool(C, D)$ is constructed. In next Section 5.3, we apply our path searching algorithm on such pruned graph to find the optimal path.

5.3 Path Searching Algorithm

Given a pruned GLORIA graph G, Algo 3 takes in a GLORIA graph G and returns the optimal path $\bar{P}$ with minimum weight as an edge list . Besides $\bar{P}$, it maintains the current minimum weight of paths $minPathWeight$.

Three utility algorithms are leveraged, GETPATHS, EDGEPRUNE and REVERSEEDGES. GETPATHS returns all the paths in the graph. By simply applying GETPATHS, one could find all paths in the GLORIA graph and selects the optimal one. However, the number of paths could be exponential in the number of edges, due to multiple outgoing edges from a node. Therefore, we leverageEDGEPRUNE and REVERSEEDGES to reduce the number of paths to linear and then find the optimal among all candidate paths. EDGEPRUNE applies *Edge pruning principle* to a given node. REVERSEEDGES reverses all edges in the GLORIA graph G.

Algorithm 1 GLORIA PATHSEARCH(G)

```
Input: GLORIA graph $G$
Output: The optimal path $\bar{P}$
1:  $\bar{P} \leftarrow \varnothing$, $minPathWeight \leftarrow +\infty$
2:  if $G.getEndNodes().size = 1$ then
3:      // Case 1: one end node
4:      $endNode \leftarrow G.getEndNodes().get(0)$
5:      EDGEPRUNE($endNode$)
6:      $\bar{P} \leftarrow$ GETPATHS$().get(0)$
7:  else
8:      $onePath \leftarrow$ true
9:      for each $endNode \in G.getEndNodes()$ do
10:         if $endNode.getIncomingeEdges().size > 1$ then
11:             $onePath \leftarrow$ false
12:     if $onePath$ = true then
13:         // Case 2: multiple end nodes with only one path
14:         $\bar{P} \leftarrow$ GETPATHS$().get(0)$
15:     else
16:         // Case 3: multiple end nodes with multiple paths
17:         REVERSEEDGES($G$)
18:         for each $n \in G$ do
19:             EDGEPRUNE($n$)
20:         for each $path \in$ GETPATHS() do
21:             if $path.w < minPathWeight$ then
22:                 $\bar{P} \leftarrow path$
23:                 $minPathWeight \leftarrow \bar{P}.w$
24:             else continue
25: return $\bar{P}$
```

GLORIA PATHSEARCH. The main algorithm GLORIA PATHSEARCH performs in following three cases.

Case 1: One end node (Line 3-6). If there is only one end node in the graph as in Figure 7(b), the optimal path can be selected by simply applying *Edge pruning principle* to the end node.

When GLORIA graph has multiple end nodes, Line 8-11 detect how many paths exist in the graph. If each end node only has one incoming edge, there is only one path existing in the graph, otherwise, there are multiple paths.

Case 2: Multiple end nodes with only one path (Line 13-14). If there is only one path existing in the graph, GETPATHS returns that path and assign it to $\bar{P}$.

Case 3: Multiple end nodes with multiple paths (Line 17-24). Since a path ends with all end nodes, each combination of incoming edge of end nodes forms a path. Assume four end nodes have two incoming edges each. Then, the total number of possible paths is 16. However, such exponential number of paths can be reduced by applying our *Edge pruning principle* reversely. Recall that by applying *Edge pruning principle*, for each node n, we only maintain one path from all start nodes to itself by computing $n.d_s$. With the full graph, we can also maintain one path from all end nodes to n by computing its distance to all end nodes $n.d_e$. Therefore, Line 17 reverses all edges in G. Line 18-19 prune the incoming edges for each node to maintain its minimum d_e. After that, for each node n, there is only one GLORIA path that passes this n. Let N be the number of nodes in G, the number of possible paths is reduced to $O(N)$. Line 20-24 enumerate all paths and selects the optimal one with minimum path weight. At last, Line 25 returns the optimal path $\bar{P}$.

Algorithm 2 REVERSEEDGES(G)

Input: GLORIA graph G

Output: G with reversed edges

```
1: for each $edge \in G$ do
2:     $leftNode \leftarrow edge.getLeftNode()$
3:     $rightNode \leftarrow edge.getRightNode()$
4:     $edge.setRightNode(leftNode)$
5:     $edge.setLeftNode(rightNode)$
6: return $G$
```

Algorithm 3 GETPATHS(G)

Input: A GLORIA graph G

Output: The list $\mathcal{P}$ of paths

```
1: $endNode \leftarrow G.getEndNodes().get(0)$
2: for each $endEdge \in endNode.getIncomingEdges()$ do
3:     $p \leftarrow \varnothing$
4:     $node \leftarrow endEdge.getLeftNode()$
5:     $node.visited \leftarrow True$
6:     $p \leftarrow p \cup node.getPath()$
7:     for each $edge \in node.getPath()$ do
8:         $nodeInPath \leftarrow edge.getLeftNode()$
9:         if $nodeInPath.getOutGoingEdges().size > 1$ then
10:            $p \leftarrow p \cup DFS(nodeInPath)$
11:    $\mathcal{P} \leftarrow \mathcal{P} \cup p$
12: return $\mathcal{P}$
```

REVERSEEDGES. This utility algorithm reverses the direction of every edge in the graph.

GETPATHS. This utility algo finds all paths in the GLORIA graph, under the assumption that each node, except for the end nodes, is only passed by one path. Line 1 starts with random end node. Each incoming edge to the end node represents a path. Therefore, line 3 initiates such path. Line 4-6 obtains the left node of the edge and get its path. Since such path maybe not complete. Line 7-10 finds the complete path passing this node. Line 9 checks that if a node in the path has multiple out going edges, it has a branching path. By applying DFS, Line 10 complete the branches and add them into the temporary path p. Then this path p is added to the path list $\mathcal{P}$ and $\mathcal{P}$ is returned.

Complexity Analysis. Given a GLORIA graph G with N nodes and T edges, utility algorithms GETPATHS, EDGEPRUNE and REVERSEEDGES are all bounded by $O(N + T)$. In Cases 1 and 2, finding the optimal path takes $O(max\{N, T\})$. In Case 3, the complexity of reversing edges and edge pruning in Lines 17-18 is

$O(N+T)$. Since each node only maintains one path, the complexity of enumerating all paths in Lines 20-26 is $O(N)$. Putting it all together, the complexity of Case 3 is $O(2N+T)$. Therefore, the complexity of GLORIA PATHSEARCH is $O(2N+T)$.

6 GLORIA Optimizer for Kleene Plus

Based on the foundation of GLORIA optimizer in Section 5, we now introduce further optimization techniques for Kleene closure.

Given a Kleene sub-pattern SEQ$(A, B)+$ for a workload $Q = \{q_1, q_2\}$ as Figure 10(b), the Kleene plus operator introduces a transition from B to A which is called ***feedback Kleene transition***. During the execution, Kleene closure matches arbitrary long event trends with exponential time complexity in the number of matched events. Thus, we focus on optimizing Kleene closure in this section. To this end, we isolate the cycle from other parts of the template and build a sub-graph for it, called ***Kleene sub-graph***. The Kleene sub-graph is concatenated to the GLORIA sub-graph of preceding sub-patterns and further pruned in the context of the whole graph. If concatenation is needed, we prune the Kleene sub-graph considering the concatenation. Therefore, instead of finding local optimality for only the Kleene sub-graph, we optimize for the graph after concatenation. Then this graph can be extended for subsequent SEQ or Kleene sub-patterns if any.

6.1 Flat Kleene Patterns

According to our GLORIA graph model, two nodes from consecutive pools are connected. In Figure 10(b), there are edges from nodes in $Pool(A, B)$ to nodes in $Pool(B, A)$ and vice versa in the template. These edges create a cycle in the Kleenesub-graph, which corresponds to sharing plans on each transition in the cycle. Figure 8(b) shows an example path that have different sharing plans, where $tran(A, B)$ has sharing plan $(1, 2)^A$ and $tran(B, A)$ has the sharing plan $(1, 2)^B$. The following Lemma 6.1 proves that such path doesn't need to be generated.

Lemma 6.1. *A cycle path that has nodes with different sharing plans can be safely pruned.*

Proof. Since we optimize the Kleene sub-pattern independently, the decision for *Q-sets* on each transition in the cycle is just either share all queries or not share at all. Let n_k and n_m be two nodes of pools on path P, we prove that $n_k = n_m$. If n_k and n_m are both sharing Q with snapshots of different event types E_0 and E_1. Assume $|E_0| > |E_1|$, according to our *Node generation principle (Local Snapshot)*, there is a path from n_k to n_m with potential sharing benefit since n_k is using more snapshots than n_m. However, since P is a cycle, there is a path from n_m to n_k too. According to *Node generation principle (Local Snapshot)*, increasing the number of snapshots from $|E_1|$ to $|E_0|$ with evaluation cost brings no sharing benefit. Therefore, $P.w$ must be larger than an alternative $P'.w$ that all nodes on P' shares Q with E_1.

Similarly, if n_k or n_m is not shared, according to Lemma 5.1, breaking a *Q-set* brings no sharing benefit, if the paths from n_k to n_m and n_m to n_k exist, $P.w$ is larger than an alternative $P'.w$ that P' shares all the time. Therefore, P doesn't need to be generated if two nodes on it are different sharing plans.

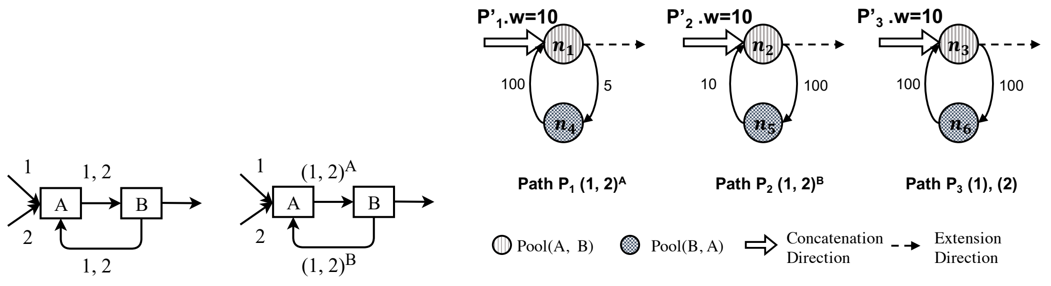
□

Based on Lemma 6.1, we propose our path generation principle.

Path Generation Principle. Given a flat Kleene sub-pattern SEQ$(E_0, \ldots, E_k)+$ in the template, its GLORIA sub-graph has $k + 1$ pools with k pools of SEQ $Pool(E_i, E_{i+1})(0 \leq i < k)$ and one pool of Kleene feedback $Pool(E_k, E_0)$. A path P in the sub-graph is a cycle that contains one node from each pool. P is generated only when all nodes in it are sharing the same *Q-set* with the same event type $E_i(0 \leq i \leq k)$ for snapshots. Otherwise, all nodes are not shared.

Node	Sharing Plan	d_s	Pruned
n_1	$(1, 2)^A$	115	Keep
n_2	$(1, 2)^B$	120	**Pruned**
n_3	$(1), (2)$	210	Keep

Table 4: d_s of nodes



(a) Template (b) Counter Example of Path Generation Principle (c) Paths in GLORIA sub-graph

Figure 8: Gloria sub-graph of flat Kleene pattern for $Q = \{q_1, q_2\}$
Stream statistics: $|A| = 5$, $|B| = 10$

Example 6.2. *Figure 8(c) shows all generated paths following Path generation principle. P_1 traverses $n_1 \in Pool(A, B)$ and $n_2 \in Pool(B, A)$, both sharing Q-set$(1, 2)$ with snapshots of A. P_2 shares Q-set$(1, 2)$ with snapshots of B and P_3 chooses to not share. The incoming edges of n_1, n_2, n_3 indicate the paths P'_1, P'_2, and P'_3 from the preceding* GLORIA *graph, associated with their respective weight. The outgoing edges of n_1, n_2, and n_3 indicate the extension direction of the Kleene sub-graph. Each edge is labelled by its weight per our cost model in Equation 3 and 4. By adding the weights of edges in the path, we have $P_1.w = 105$, $P_2.w = 110$ and $P_1.w = 200$.*

With the weight of path in the Kleene sub-graph as well as the weight of path in the preceding GLORIA graph, we can prune the nodes that are expensive for extension.

Pruning. For each path P_i, there is a path P'_i from the concatenation direction, and a node n_k as the source of the extension direction. We update the $n_k.d_s$ as follows:

$$n_k.d_s = P'_i.w + P_i.w \tag{24}$$

By applying *Node pruning principle* on the source nodes, the optimizer selects the optimal path with minimum weight for future graph extension, considering the concatenation.

Example 6.3. *Continuing Example 6.2, assume $Q = \{q_1, q_2\}$ that both q_1, q_2 start with A, then n_1–n_3 is concatenated to a common start node n^Q_{st}. By applying the cost model in Equation 2, P'_1 to P'_3 have the same weight 10. As the source nodes of extension, $n_1.d_s = 115$, $n_2.d_s = 120$ and $n_3.d_s = 210$. According to Node pruning principle, n_2 is pruned compared with n_1, together with the whole path P_2.*

6.2 Nested Kleene Patterns

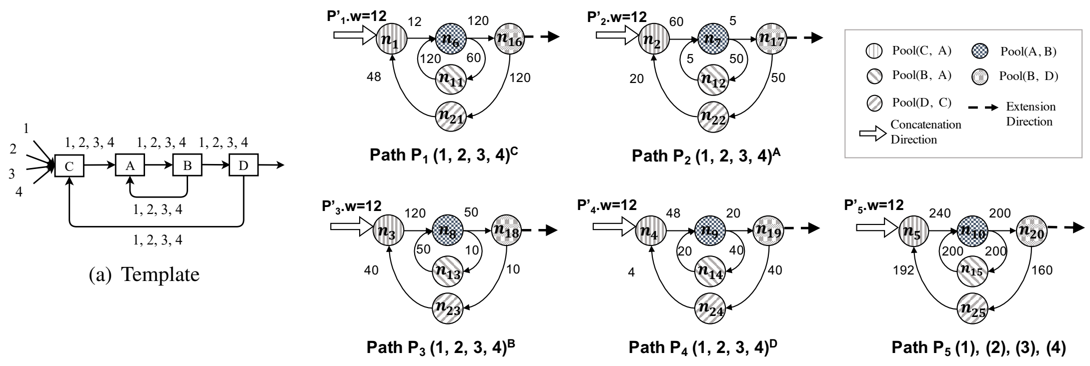
Nested Kleene patterns introduce nested cycles in the template. Figure 9(a) shows the partial template of the nested Kleene sub-pattern $\mathsf{SEQ}(C, \mathsf{SEQ}(A, B)+, D)+$ of a workload $Q = \{q_1, q_2, q_3, q_4\}$. We now prove that the *Path generation principle* still applies to arbitrarily nested Kleene patterns.

Lemma 6.4. *Path generation principle applies to the Kleene sub-graph of an arbitrarily nested Kleene sub-pattern.*

Proof. The path of the nested Kleene sub-pattern is a nested cycle path. According to Lemma 6.1, every single cycle path in the nested cycle path has the same sharing plan for each node, therefore, the nested cycle path has the same sharing plan for every node on it. □

Based on Lemma 6.4, the sub-graph of a nested Kleene sub-pattern can be constructed and pruned in the same way as the flat Kleene sub-patterns by applying *Path generation principle* and *Node pruning principle*.

Example 6.5. *Figure 9(b) shows all paths of the nested Kleene sub-pattern in Figure 9(a). According to Path generation principle, five paths are generated where P_1–P_4 correspond to sharing Q with snapshots of different event types C, A, B, D respectively, and P_5 corresponds to non-sharing. Each edge is labelled by its weight per our cost model. We add the weight $P_i.w$ of the path in current sub-graph and the weight of the path $P'_i.w$ from the concatenated sub-graph. Each node n_{16}–n_{20} obtains its d_s shown in Table 5. We apply Node pruning principle to the nodes in $Pool(B, D)$ and prune n_{16}–n_{18}.*



(a) Template

(b) Paths in GLORIA sub-graph

Figure 9: GLORIA Kleene sub-graph of nested Kleene sub-pattern for $Q = \{q_1, q_2, q_3, q_4\}$
Stream statistics: $|A| = 5$, $|B| = 10$, $|C| = 12$, $|D| = 4$

Node	Sharing Plan	d_s	Pruned
n_{16}	$(1-4)^C$	492	**Pruned**
n_{17}	$(1-4)^A$	252	**Pruned**
n_{18}	$(1-4)^B$	292	**Pruned**
n_{19}	$(1-4)^D$	140	Keep
n_{20}	$(1), (2), (3), (4)$	1204	Keep

Table 5: d_s of nodes

Path Search. Given that paths in the Kleene sub-graph are cycles, they require minor modification to only GETPATHS without affecting the main algorithm GLORIA PATHSEARCH (Algorithm 3). Specifically, Case 1 in GLORIA PATHSEARCH remains the same since EDGEPRUNE only prunes the multiple incoming edges for a node which won't happen to nodes in the Kleene sub-graph. In case 2, since there is only one path, the path can be directly output even with a cycle in it. Case 3 requires that one node is only passed by one path which is exactly what our Kleene sub-graph provides. Thus, GLORIA PATHSEARCH stays the same. We modify GETPATHS to accept cycles in a path. Also, as the template captures the structure of the path, GETPATHS won't fall into an infinite loop.

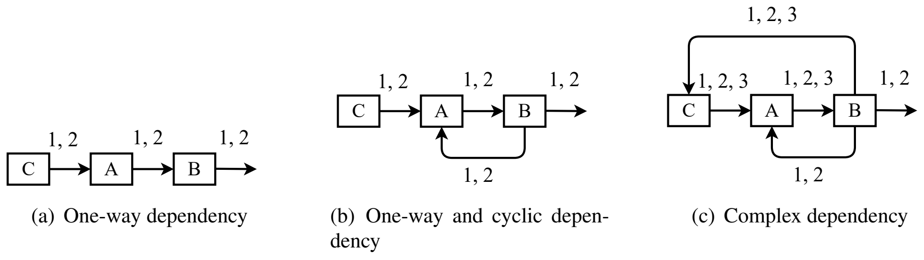
Optimality Discussion. GLORIA optimizer only considers either sharing all queries or not sharing at all for Kleene sub-pattern. Thus, when the sub-graph of the Kleene sub-pattern is concatenated to the whole graph, we may omit the opportunities of keeping certain *Q-set*s in the concatenated sub-graph, which may sacrifice optimality. However, in a special case when all queries start with the Kleene sub-pattern, the template starts with the cycle, no existing *Q-set*s need to be considered, and GLORIA finds the optimal path for the sub-graph of the Kleene pattern.

7 Discussion

In this section, we sketch out how GLORIA can be extended to handle SEQ vs. Kleene sub-parts, dynamic optimization, configurable cost models and additional aggregation functions.

7.1 SEQ vs. Kleene sub-parts.

To maximize the sharing benefit, GLORIA generates a pool based on the preceding pool (Section 5.1). Such sequential optimization considers the long-term benefit but however introduces optimization dependencies. In a template only contains SEQ sub-patterns as in Figure 10(a), such dependencies are one-way. The generation of $Pool(A, B)$ is only dependent on $Pool(C, A)$. However, with the join of the Kleenesub-patterns as in Figure 10(b), not only we have the one-way dependencies, we also have a new cyclic dependency. The generation of $Pool(A, B)$ is not only dependent on $Pool(C, A)$ but also dependent on $Pool(B, A)$, which is vice versa dependent on $Pool(A, B)$. This introduces two problems. First, the optimizer needs to optimize with both one-way and cyclic dependencies. Second, even only with the cyclic dependencies, there is a chicken-and-egg problem in respect with optimization. To solve these problems, the GLORIA optimizer first isolates the



(a) One-way dependency (b) One-way and cyclic dependency (c) Complex dependency

Figure 10: Optimization dependencies

Kleenesub-patterns with the SEQ sub-patterns. Then it focuses on optimizing the Kleenesub-pattern as a whole to solve the cyclic dependencies (Section 6). To achieve this goal, we make assumptions in Section 6 that a Kleenesub-pattern can only be shared by a workload Q if all $q \in Q$ contains it. This assumption allows the isolation between the SEQ and Kleenesub-patterns, as well as the isolation of different Kleenesub-patterns. Based on this assumption, GLORIA optimizer finds a consistent sharing plan for the flat or nested Kleenesub-pattern, which always satisfies the cyclic dependencies.

Without the above assumptions, sharing can potentially happen to any overlapping part among arbitrary SEQ, flat or nested Kleenesub-patterns, which introduces really complex dependencies as in Figure 10(c). Except for the one-way dependencies indicated by the straight arrows, two overlapping cyclic dependencies exist as for the longer Kleenesub-pattern SEQ$(A, B, C)+$ and the shorter Kleenesub-pattern SEQ$(A, B)+$. Optimization in this case could be complicated by the fact that no one-way dependency or a cyclic dependency can be optimized independently. This problem will be even harder with the nested Kleenesub-patterns. Thus, we leave this as our future work.

7.2 Dynamic Optimization

Dynamic optimization at run-time has been well-studied in the stream processing literature [23, 40, 18]. However, GLORIA has its unique challenges with respect to dynamic optimization, when the setting changes, including either the query workload or the data stream statistics.

Challenge 1. GLORIA 's progressive pruning during the graph construction significantly reduces the size of GLORIA graph in the early stage. However, this in turn makes it harder to incrementally recover the GLORIA graph for the purposes of incrementally adapting the GLORIA graph. When the workload changes significantly, the originally pruned edges or nodes of a pool could become suboptimal in the new setting, so that the pool needs to be re-constructed.

Challenge 2. The sequential node generation process from pool to pool limits the number of nodes in each pool - which is a plus. However, it can also can trigger a chain reaction when one pool is changed due to the dynamic workload. To obtain a new GLORIA graph adapted to the new setting, such changes need to be propagated to all the subsequent pools. In the worst case, a from scratch re-optimization may be required.

In summary, to guarantee the performance quality of GLORIA , a heavy re-optimization from the scratch is required. However, to quickly respond to dynamic changes, a lightweight re-optimization strategy is needed. Specifically, when new queries are added or removed, the executor needs an optimized workload sharing plan for the current queries. When the data statistics change, the executor needs a workload sharing plan that performs better than the current one, if any.

We sketch a lightweight solution as follows that potentially can be used to address the above problems. A full-fledged solution trading off between the real-time responsiveness and the optimization quality of the resulting plan is an open question which is beyond the scope of this paper.

With respect to dynamic workloads, the GLORIA optimizer can incrementally optimize for the newly added queries. The GLORIA optimizer generates a separate GLORIA graph for the query, and searches the optimal path for the newly added queries. After that, the sharing plan for the newly added queries is added into the executor, running together with the current sharing plan. Since each sharing plan guides the execution for a unique set of queries, different sharing plans will not conflict with each other. Such incremental optimization sacrifices the sharing opportunities for the new queries with the existing ones, in trade of a fast response for the new queries. When a query is removed, the GLORIA optimizer removes that query from the current sharing plan and feeds the new sharing plan to the executor. For *Q-set*s that contain only one query after the removal, the GLORIA optimizer converts that *Q-set* into a singleton query without sharing.

With respect to dynamic statistics, sampling techniques [24] can be leveraged to sample data statistics and monitor fluctuations. The GLORIA optimizer reacts to significant statistical changes that exceed a threshold set by the administrator. To quickly adapt to the changes, GLORIA optimizer keeps the pruned GLORIA graph as shown in Figure 7(b). Let E_j be an event type with changed statistics, the GLORIA optimizer updates the weight of relevant edges for all relevant pools $Pool(E_i, E_j)$ and $Pool(E_j, E_h)$, as well as the weight of relevant incoming and outgoing edges of the nodes in which E_j works as an event type of snapshots. Let N be the number of nodes and T be the number of edges in the GLORIA graph, the complexity of such weight update is $O(T)$ in the worst case. With the same GLORIA graph structure and the different weights, the path search algorithm searches the optimal path for the updated GLORIA graph in $O(2N + T)$. Together, the complexity of this re-optimization process is thus efficient, namely, $O(2N + 2T)$.

Plan Migration. Once a new workload sharing plan has been generated, the executor needs to migrate from the old sharing plan to the new one. Existing works [40] provide strategies for such plan migration, which involves mapping and copying states for initialization from the existing plan to the new one. While these methods can largely be adopted, attention must be paid to the handling of snapshot expressions as detailed below. In a nutshell, during the migration, when an event e arrives, our executor computes the snapshot expressions following the new sharing plan. Since the predecessors *Q-set*s for event e could be different in the old and new sharing plans, when visiting the predecessors of e and those carry expressions for the old sharing plan, the executor first computes the expression for e by summing up the expressions of the predecessors. It then materializes these expressions into result values for each query that can be sent along the new sharing plan. Once all existing events use the expressions according to the new sharing plan, the plan migration is finished.

Reconfiguration Costs. Based on our understanding, the reconfiguration costs cover the costs of the re-optimization and of the plan migration. To better adapt to dynamic optimization, the optimizer is running independently on a separate machine. Therefore, only the cost of plan migration affects the reconfiguration. The cost of the plan migration is dominant by the cost of expression evaluation as we discussed above, for two reasons. First, for the aggregate values that need to be output during or even after the plan migration, the cost of such evaluation and output is already covered by the cost of the old sharing plan. Second, for the events that are applying the new sharing plan, the execution cost is covered by the cost of the new sharing plan. Therefore, the only extra cost of the plan migration is introduced by the extra expression evaluation for compatibility. In the worst case, the old and sharing plans are totally different so that the expression evaluation is required for every existing event. Given a workload Q, let n be the number of existing events, for each event, the length of its snapshot expression is bounded by $O(n)$. Therefore, the reconfiguration cost is bounded by $O(n^2 \times |Q|)$.

Intuitively, we make the assumption that the changes are relatively stable. For dynamic workloads, the optimizer always migrates to the new sharing plan to support the new workload. For dynamic statistics, when the long-term benefit, which can be computed by the difference of the costs of the old and new sharing plans, exceeds the reconfiguration costs, the optimizer triggers the plan migration.

7.3 Configurable Cost Model

Our response sketching strategies for handling changes in the cost model is broken into two parts. The first part concerns the optimization on the GLORIA graph (once established), with the weights on the GLORIA graph derived based on the underlying cost model. This is at the core of the GLORIA optimizer and focuses primarily on the edge pruning rule (Section 4.3) and the path search algorithm (Section 4.4). The second part concerns the initial construction of the GLORIA graph, transforming the plan optimization problem into a GLORIA graph path finding problem. This second part thus covers the rules to keep the size of the graph compact during the actual construction process. It includes the node generation rules (Section 4.2) and node pruning rule (Section 4.3).

The first part, which is the core of the GLORIA optimizer, is independent of the cost model since it simply prunes and searches based on the weighted GLORIA graph structure. While a different cost model may change the weights associated with the edges, it would not affect the correctness of the graph pruning and graph search algorithms. In particular, the edge pruning rule prunes the expensive paths passing through a node. Lemma 5.8 proves that the edge pruning rule does not discard the optimal path.

The path search algorithm searches for the optimal path with the minimal weight. As long as the weighted GLORIA graph can be constructed, the edge pruning rule and path search algorithms thus hold, regardless of the cost model deciding which weights to place on the edges.

The second part holds independently of the cost model under the following assumptions: 1. the shared execution costs are proportional to the length of the snapshot expressions, and 2. the shared execution costs are proportional to the number of *Q-set*s. Specifically, the *Node generation principle (Reuse)* always holds since it avoids the expensive expression evaluation. However, the *Node generation principle (Local Snapshot)* holds under the first assumption. If the snapshot expressions can be shortened by replacing the old snapshots by local snapshots of a less frequent event type, even though the evaluation is expensive, this would reduce the shared execution costs of the succeeding event type. Based on the second assumption, the *Node generation principle (Merging)* holds since by merging multiple *Q-set*s into one single *Q-set*s, the number of computations for *Q-set*s is reduced to 1. With respect to the *Node pruning principle*, under the first assumption, given two nodes n_m and n_k with the same *Q-set*s, the node n_m with more snapshots introduces more execution costs compared with n_k. Based on that, Lemma 5.11 proves the correctness of *Node pruning principle* that n_m can be safely pruned.

7.4 Complex Aggregation functions

Targeting online incremental aggregation, the GLORIA optimizer supports aggregation functions that are associative and commutative. Specifically, associativity allows aggregation be computed using partial aggregates that subsequently can be composed into the final aggregate value. This is at the core of the vast majority of the online aggregation methods in both traditional databases [16] and streaming systems [21, 28, 30, 31]. Meanwhile, commutativity implies that aggregation can ignore the order of the input, thus relaxing the assumption of the stream order.

There is rather limited work on supporting other complex aggregation functions like QUANTILE or ARGMAX (that are not both associative and commutative) [12, 36]. These systems design special-purpose execution strategies for aggregation. The simplest and most common among them is to store all tuples to be aggregated within the pane (window) instead of summarizing them up into compact partial aggregates.

However, for streaming systems, storing all tuples risks to cause the operator state to grow prohibitively large. To avoid this state explosion, Flink [3] provides a parameter that controls the size of the state. That is, when the state reaches the maximal allowed size as indicated by the parameter, tuples are selectively dropped from the state (shed).

This practical approach, while simple, can affect the correctness of the aggregation results [4]. That is, the aggregate result would become a mere approximation. More sophisticated optimization custom for specific aggregation support of non-associative/non-commutative functions is a largely open problem in the literature, that we consider to be orthogonal to the focus on our work here. We thus leave this for future work.

8 EXPERIMENTAL EVALUATION

8.1 Experimental Setup

Environment. We implemented GLORIA in Java with OpenJDK 16.0.1 on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. Our code is available online [9]. Each experiment is reported by the average of 15 runs.

Data sets. We evaluate GLORIA using three real-world data sets.

• *NASDAQ Stock data set (Stock)* [5] contains the stock price history of 20 years. Each record represents a primitive event which carries a company identifier, a timestamp in minutes, the open and close price, the highest and lowest price as well as the trading volume. There are 3258 unique company identifiers which are used as event types.

• *New York City Taxi data set (Taxi)* [8] contains 2.63 billion yellow taxi trip records in New York City in 2019-2020. Each record is an event that carries a timestamp in seconds, vendor id, pick-up and drop-off location identifier, a passenger number, the trip distance and the total price. There are 217 unique pick-up locations which are used as event types.

• *Dublin Bus GPS data set (Bus)* [1] consists of GPS records of buses in Dublin which was collected by Dublin City Council in a month period in 2013. Each record is a tuple of a timestamp in microseconds, a line id, a vehicle journey id, a congestion indicator, the coordinates and a delay time. We use the vehicle journey id as the event type which has 4368 unique values.

Event Trend Aggregation Queries. To evaluate the effectiveness of GLORIA on different query workloads, we generate three types of workload on each data set.

• *SEQ workload* focuses on SEQ patterns. Queries in this workload has different shareable SEQ patterns, group-by, predicates, as well as aggregates (e.g., COUNT(*), AVG, SUM, etc.). Window sizes are powers of 5 minutes. Windows slide every 5 minutes.

• *Kleene workload* consists of queries with one shareable flat or nested Kleene sub-pattern as well as different SEQ sub-patterns. The length of shareable Kleene sub-patterns ranges from 2 to 10 and the number of nested Kleene sub-patterns ranges from 1 to 5. The group-by, predicate, window and aggregate settings are the same as SEQ workload.

• *Mixed workload* is introduced to evaluate how our GLORIA performs on a real-world workload, where the above SEQ and Kleene queries appear in one workload. The ratio of Kleene and SEQ queries in the mixed workload ranges from 1:6 to 1:2. The group-by, predicates, window and aggregate settings are the same as SEQ workload.

Methodology. We evaluate both the GLORIA Optimizer and the execution of the produced sharing plan. We compare our GLORIA Optimizer to two optimization approaches.

• *Greedy Optimizer (Greedy).* For each pool, the *Greedy* optimizer only considers the transition sharing plan with the minimum incoming edge weight (Section 5.1), thus potentially missing sharing benefits. Specifically, with respect to node generation, it either applies *Node generation principle (Reuse)* for sharing, or chooses not to shared, based on the incoming edge weight. As it only generates one node for each pool, the *Greedy* optimizer simply returns the workload sharing plan directly without pruning or path searching.

• GLORIA *Optimizer without pruning (NoPrune).* To evaluate the *effectiveness* of pruning rules, we compare the GLORIA Optimizer against an optimizer without pruning rules (Section 5.2). It generates a full GLORIA graph as shown in Figures 7(a) and 9(b), which is prohibitively expensive to construct. After the construction, the path search algorithm generates an optimized workload sharing plan.

We also evaluate the *execution costs of the optimized sharing plans* generated by GLORIA optimizer by comparing those with the following metthods:

• *Sharing plan of Greedy Optimizer*.

• GRETA[29], a state-of-the-art method supporting online aggregation over nested Kleenepatterns.

• HAMLET[28], a state-of-the-art shared online aggregation method equipped with a dynamic optimizer for flat Kleenepatterns on bursty streams.

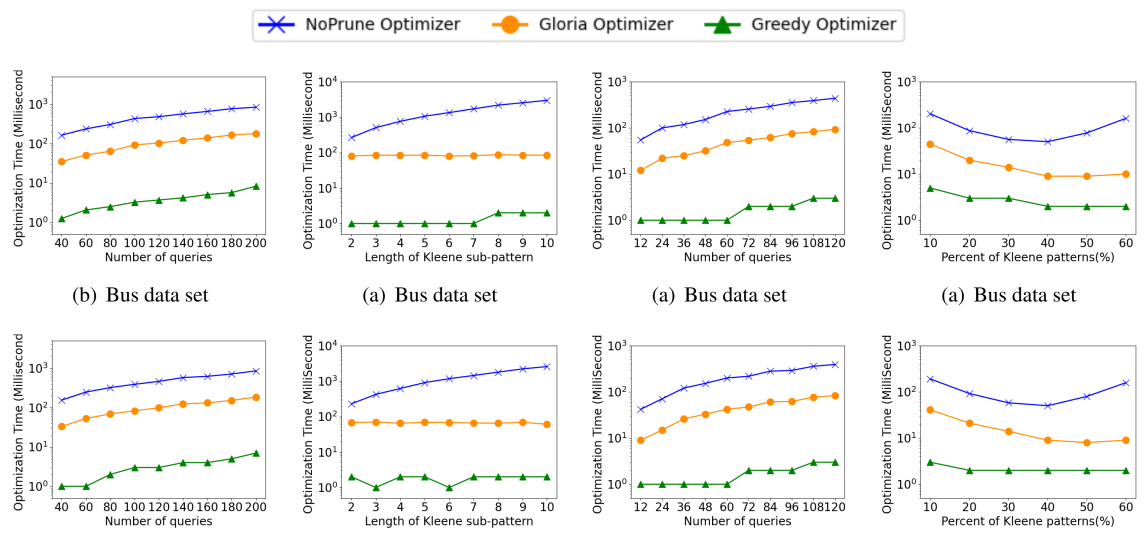
Metrics. We measure the *Optimization Time* in milliseconds as the average time difference between the time of receiving the input workload and the time GLORIA producing the sharing plan. This includes the duration of template construction, graph construction, and path searching. *Peak Memory* corresponds to the maximal memory consumed during graph construction and path searching. For query execution, we use *Latency* in seconds as the average time difference between the time of producing aggregation results for a query in the workload and the arrival time of the last relevant event. *Throughput* is measured as the average number of events processed by all queries per second.

8.2 GLORIA Optimization Evaluation

SEQ Workload. To evaluate the effectiveness of pruning rules on SEQ patterns, we measure the optimization time of three optimizers on *Bus* and *Taxi* data sets in Figure 11 while varying the number of queries in the SEQ workload from 40 to 200. On both data sets, GLORIA optimizer consistently outperforms the *NoPrune* optimizer by a factor of 5 to 7, and is slower than the *Greedy* optimizer by 1.2 order of magnitude. Even though the *Greedy* optimizer is the fastest among the three, it cannot guarantee the quality of the selected sharing plan. In contrast, both *NoPrune* and GLORIA optimizer return the optimized sharing plan for the given workload. Comparing to *NoPrune*, GLORIA pruning rules reduce the optimization time significantly. Such time saving comes from both graph construction and path searching. During the graph construction, the pruning rules reduce the number of nodes in a pool, which in turn reduces the number of generated nodes in the succeeding pools (Section 5.2). Given that the resulting graph is much smaller, the path searching runs faster to find the optimal path. In summary, GLORIA optimizer can efficiently produce a sharing plan with the optimality guarantee.

Kleene Workload. To evaluate the effectiveness of pruning rules on Kleene patterns, we compare three optimizers on the Kleene workload with 100 queries in Figure 12 while varying the length of Kleene sub-patterns from 2 to 10. GLORIA consistently outperforms *Greedy* optimizer and *NoPrune* optimizer. Since *Greedy* optimizer only maintains one node in each pool, it is the fastest among all optimizers. As the length of Kleene sub-patterns increases, the performance difference between GLORIA and *NoPrune* increases from 2-fold to 13-fold.

Such performance difference is primarily due to the path consistency property introduced in Lemma 6.1. It ensures that the increasing length of Kleene pattern does not increase the number of cycle sharing plans. Hence for a given Kleene workload, the size of the generated sub-graph for Kleene sub-patterns is always small, leading to negligible optimization time increase. In contrast, the *NoPrune* optimizer does not prune any candidate cycle sharing plans. Even though the number of cycle sharing plans is growing linearly, with the



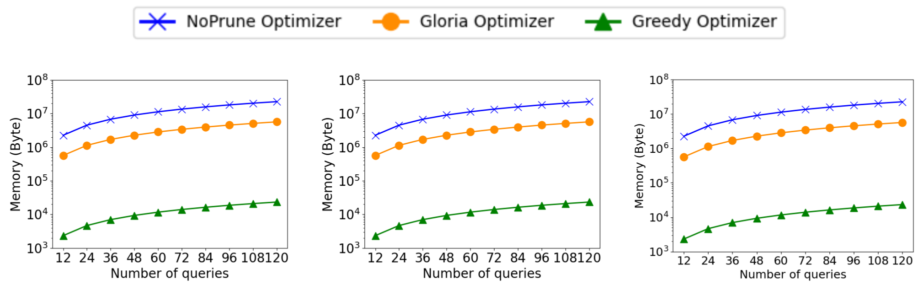
(b) Bus data set (a) Bus data set (a) Bus data set (a) Bus data set

(c) Taxi data set (b) Taxi data set (b) Taxi data set (b) Taxi data set

Figure 11: Varying # queries (SEQ) **Figure 12:** Varying length of Kleene (Kleene) **Figure 13:** Varying # queries (Mixed) **Figure 14:** Varying percent of Kleene queries (Mixed)

following SEQ sub-patterns, the size of GLORIA graph could still grow exponentially in the worst case. This is consistent with the SEQ workload. Pruning expensive nodes early on prevents the graph from exploding, which benefits both graph construction and path searching.

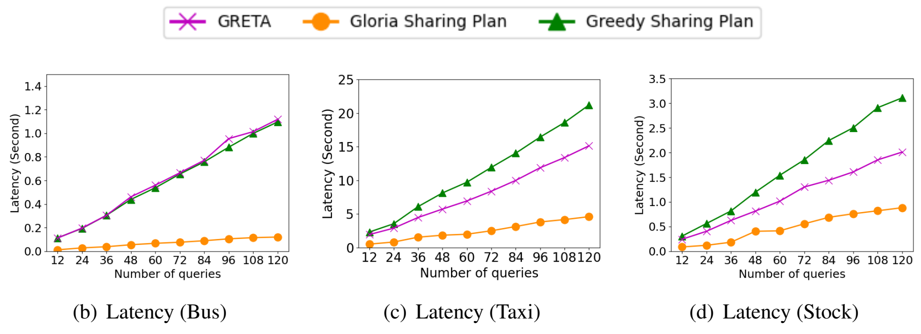
Mixed Workload. Figure 13 compares the three optimizers on mixed workloads with a SEQ-to-Kleene ratio as 6:1 with varying number of queries. Again, GLORIA optimizer constantly outperforms *NoPrune* optimizer by 5-fold and is slower than the *Greedy* optimizer by 1 order of magnitude. In Figure 14, we compare the three optimizers on a mixed workload with 100 queries. We vary the percentage of Kleene queries from 10% to 60%. As the number of Kleene queries increases, the GLORIA optimizer outperforms *NoPrune* optimizer by 6-fold to 1.2 order of magnitude. More precisely, when the percentage of Kleene queries increases from 10% to 40%, the optimization time of *NoPrune* optimizer drops since fewer SEQ queries are shared as well as the limited number of Kleene queries. When the percentage of Kleene queries is larger than 40%, the optimization time is dominated by the cost of optimizing the shared Kleene queries. In contrast, thanks to the pruning rule applied to the Kleene sub-graph (Section 6), the optimization time of GLORIA optimizer decreases as the percent of Kleene queries increases. Figure 15 compares the memory consumption of the three optimizers. Due to the pruning rules, GLORIA optimizer only consumes 25% memory of the *NoPrune* optimizer on both data sets.



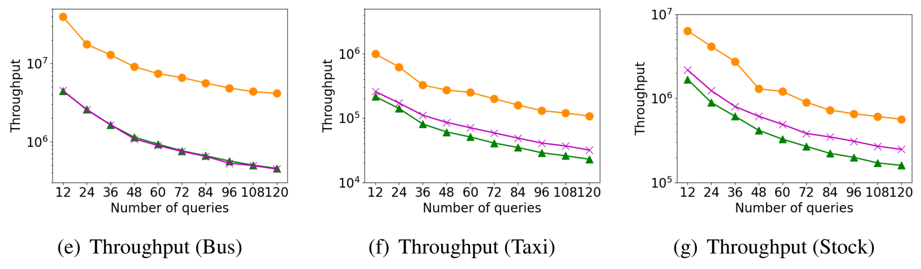
(b) Bus (c) Taxi (d) Stock

Figure 15: Memory Consumption (Mixed)

To evaluate the quality of the sharing plan produced by GLORIA , we evaluate several mixed workloads over all three data sets, shown in Figure 16. GRETA executor represents an execution plan that runs each query independently without any sharing. The *Greedy* and GLORIA sharing plans are returned by the *Greedy* and GLORIA optimizer, respectively. In this experiment, we measure the latency and throughput. We vary the number of queries in the workload from 12 to 120 with a fixed SEQ-to-Kleene ratio as 6:1.



(b) Latency (Bus) (c) Latency (Taxi) (d) Latency (Stock)



(e) Throughput (Bus) (f) Throughput (Taxi) (g) Throughput (Stock)

Figure 16: Execution with different sharing plans (Mixed)

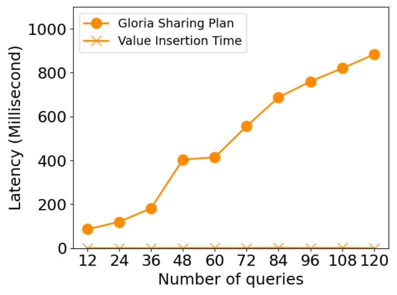


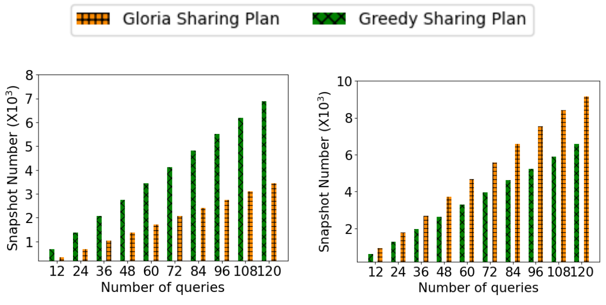
Figure 17: Value Insertion Time vs. Latency (Stock)

We measure throughput in Figures 16(e), 16(f) and 16(g). GLORIA sharing plan outperforms *Greedy* plan and GRETA by 3-fold and 10-fold, respectively. Such performance gain is due to GLORIA 's selection of sharing opportunities. Specifically, GRETA is not shared and *Greedy* optimizer fails to harvest all beneficial sharing opportunities since it aggressively reuses existing snapshots without introducing new ones even when they are beneficial. GLORIA optimizer evaluates the cost and benefits of different selection of snapshots and picks the beneficial ones. Therefore, GLORIA optimizer gets rid of expensive plans, while keeping all potentially beneficial plans. When the query number increases from 12 to 120, the execution latency of GLORIA sharing plan achieves 77-91% and 68-93% speed-up compared to *Greedy* sharing plan and GRETA, respectively.

We observe that GRETA outperforms (Figures 16(c) and 16(d)) or performs similarly (Figure 16(b)) to the execution of the *Greedy* sharing plan, which emphasizes the drawback of greedy strategy and the importance of long-term optimization. If no sharing is the local optimal for a transition, the executor with *Greedy* sharing plan performs similar to GRETA. Alternatively, if the *Greedy* optimizer selects an event type with high frequency as snapshots, these snapshots will be reused for many following transitions. In the execution, the overhead of summing long snapshot expressions could outweigh the benefit of sharing, which causes a non-beneficial sharing scenario. In contrast, GLORIA optimizer can detect this situation and apply *Node generation principle (Merging)* or *Node generation principle (Local Snapshot)* to stop the non-beneficial sharing.

To examine the claim that the cost of value insertion and snapshot maintenance is negligible, we also measure the time of these two operations on the stock data set in Figure 17. When the number of queries increases, the latency of the GLORIA sharing plan increases meanwhile the value insertion time stays under 1 millisecond. This result supports our claim about the cost of the value insertion and snapshot maintenance is negligible.

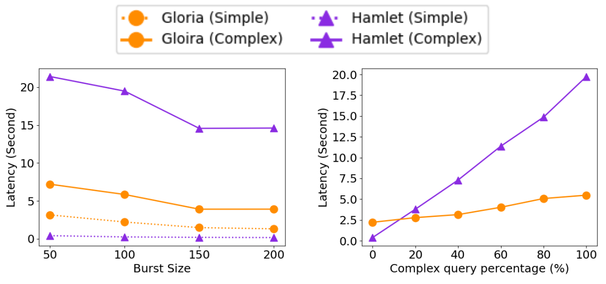
Snapshot Selections. We also evaluate the number of snapshots created by GLORIA and the *Greedy* sharing plans during the execution. An interesting observation is that GLORIA sharing plan may create more snapshots than the *Greedy* sharing plan but still outperforms the latter. Figure 18 compares the number of snapshots created by GLORIA sharing plan and *Greedy* sharing plan on *Bus* and *Stock* data set. GLORIA sharing plan creates fewer snapshots than *Greedy* in Figure 18(b)) and more snapshots than *Greedy* in Figure 18(c).



(b) Bus data set (c) Stock data set

Figure 18: Number of generated snapshots (Mixed)

The above observation is not surprising. According to our cost model in Equations 3 and 4, the cost of shared execution is not dominated by the total number of snapshots but the number of snapshots used by the events. In Figure 18(b), the *Greedy* sharing plan introduces a lot of snapshots with little sharing benefit. In this case, the sharing benefit is outweighed by the extra cost of snapshot propagation and evaluation. GLORIA optimizer factors in such non-beneficial sharing scenario and does not introduce new snapshots unless they are beneficial. In this case, GLORIA sharing plan runs faster than the *Greedy* with fewer snapshots. In Figure 18(c), the *Greedy* sharing plan introduce fewer snapshots and reuse them throughout the sharing plan. However, it misses alternative plans, in which new snapshots with a much lower frequency should be introduced. In this case, GLORIA optimizer stops reusing the existing snapshots and creates new ones with greater sharing benefits. Consequently, these new snapshots reduce the overall execution cost, even GLORIA creates more snapshots than the *Greedy* optimizer.



(a) Vary burst size (b) Vary complex query percentage

Figure 19: Gloria vs. Hamlet Execution

GLORIA VS. HAMLET. To give a fair comparison between HAMLET and GLORIA, we follow HAMLET's bursty stream assumption and create bursty streams. To examine the optimization of HAMLET and GLORIA, we generate a simple and a complex workload, which both contain 100 queries. In the simple workload, all predicates are the same. Each query is a sequence pattern of length 3, with a Kleene operator on a single event type, such as SEQ(A, B+, C). In the complex workload, each query is a flat Kleene query of length 8 with different predicates. The workloads can be found in [10].

In Figure 19(a), we measure the latency of the two methods for the simple and the complex workloads, while varying the burst sizes of the stream. With an increasing burst size, the latency of both methods decreases on both workload, because less expensive predecessor reaching happens in a bigger burst of same event type. The experiment result shows that HAMLET wins on the simple workload, but GLORIA constantly outperforms HAMLET by a factor of 3 on the complex workload. With respect to the optimization, HAMLET's dynamic optimizer only optimizes for the Kleene sub-pattern like B+, and GLORIA static optimizer analyzes all the sharing opportunities. Therefore, GLORIA outperforms HAMLET on the complex workload, since the longer patterns release the advantage of GLORIA 's static optimizer, which achieves a larger optimization scope. With respect to the execution, when all predicates are the same in the simple workload, HAMLET's shared execution processes the burst as a batch, without interruption. However, in the complex workload scenario with different predicates, HAMLET deteriorates directly to a non-sharing plan if many snapshots will be created. However, GLORIA follows the optimized sharing plan generated by its optimizer, which guarantees to be cheaper than a non-sharing plan.

We further measure the latency of these two methods on the bursty stream with burst size 100, varying the percentage of complex queries in the workload in Figure 19(b). In consistent with Figure 19(a), HAMLET outperforms GLORIA when there is no complex queries. However, with the percentage increases from 20% to 100%, thanks to the global GLORIA static optimizer, the performance difference between GLORIA and HAMLET increases from 30% to 3-fold.

9 Related Work

Complex Event Processing Systems. CEP have gained popularity in the recent years [2, 3, 6, 7]. Some approaches use a Finite State Automaton (FSA) as an execution framework for pattern matching [11, 13, 37, 38]. Others employ tree-based models [26]. Some approaches study lazy match detection [22], compact event graph encoding [27], and join plan generation [20]. We refer to the recent survey [14] for further details.

Online Event Trend Aggregation. A broad variety of optimization techniques have been introduced to minimizing processing time and resource consumption of event trend aggregation [32, 38, 30, 29]. A-Seq [32] introduces online aggregation of event sequences, i.e., sequence aggregation without sequence construction. GRETA [29] extends A-Seq by Kleene closure. Cogra [30] further generalizes online trend aggregation by various event matching semantics. However, none of these approaches addresses the challenges of multi-query workloads, which is our focus.

CEP Multi-query Optimization. Following the principles commonly used in relational database systems [35], pattern sharing techniques for CEP have attracted considerable attention. RUMOR [17] defines a set of rules for merging queries in NFA-based RDBMS and stream processing systems. E-Cube [25] inserts sequence queries into a hierarchy based on concept and pattern refinement relations. SPASS [33] estimates the benefit of sharing for event sequence construction using intra-query and inter-query event correlations. MOTTO [39] applies merge, decomposition, and operator transformation techniques to re-write pattern matching queries. Kolchinsky et al. [21] combine sharing and pattern reordering optimizations for both NFA-based and tree-based query plans. Most recently, HAMLET [28] is introduced to adaptively make sharing decisions at run time, depending on the current stream properties, to harvest the maximum sharing benefit. It is also equipped with a highly efficient shared trend aggregation strategy that avoids trend construction.

However, some of these approaches [21, 33, 39] do not support online aggregation of event sequences, i.e., they construct all event sequences prior to their aggregation, which degrades query performance. To the best of our knowledge, SHARON [31], Muse [34], and HAMLET [28] are the only solutions that support shared online aggregation. However, SHARON does not support Kleene closure, MCEP and COGRA [21, 30] only support sharing flat Kleene patterns with one single event type. HAMLET only considers sharing opportunities among a special case Kleene sub-pattern, namely, one that is flat and only contains a single event type. These assumptions often result in sub-optimal sharing plans for event trend aggregation queries, since many sharing opportunities are missed.

10 Conclusion

GLORIA introduces a graph-based sharing optimizer for event trend aggregation. We transform the sharing plan search space into a GLORIA graph and map the event trend aggregation sharing problem to a path search problem. We propose effective pruning rules to reduce the size of the GLORIA graph during its construction. We propose an efficient path search algorithm to find a high-quality sharing plan in linear time. Our experiments demonstrate that the sharing plan produced by the GLORIA optimizer achieves significant performance gains compared to state-of-the-art approaches.

References

[1] Dublin Bus. https://data.smartdublin.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insight-project/

resource/00c65697-9ed6-43cb-a2b7-9e20cf323cb3.

[2] Esper. http://www.espertech.com/.

[3] Flink. https://flink.apache.org/.

[4] Flink Docs. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/table/sql/queries/group-agg/.

[5] Historical stock data. http://www.eoddata.com.

[6] Microsoft StreamInsight. https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx.

[7] Oracle Stream Analytics. https://www.oracle.com/middleware/technologies/stream-processing.html.

[8] Unified New York City taxi and Uber data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[9] Gloria code. https://anonymous.4open.science/r/Gloria-C1D5/, 2022.

[10] Gloria vs. hamlet experiment workloads. https://anonymous.4open.science/r/Gloria-C1D5/src/main/resources/stock/GloriaVSHamletWorkload/MixWorkload, 2022.

[11] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[12] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[13] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[14] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis. Complex event recognition in the Big Data era: A survey. *PVLDB*, 29(1):313–352, 2020.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, pages 29–53, 1997.

[16] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182, 1997.

[17] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.

[18] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 607–622, 2019.

[19] M. Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, 102(1):63–87, 2003.

[20] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. In *PVLDB*, pages 1332–1345, 2018.

[21] I. Kolchinsky and A. Schuster. Real-time multi-pattern detection over event streams. In *SIGMOD*, pages 589–606, 2019.

[22] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45, 2015.

[23] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 623–634, 2006.

[24] C. Lei, E. A. Rundensteiner, and J. D. Guttman. Robust distributed stream processing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 817–828, 2013.

[25] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.

[26] Y. Mei and S. Madden. ZStream: A cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[27] O. Poppe, C. Lei, S. Ahmed, and E. Rundensteiner. Complete event trend detection in high-rate streams. In *SIGMOD*, pages 109–124, 2017.

[28] O. Poppe, C. Lei, L. Ma, and E. A. Rundensteiner. To share, or not to share online event trend aggregation overbursty event streams. In *SIGMOD*, pages 1452–1464, 2021.

[29] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. Greta: Graph-based real-time event trend aggregation. In *VLDB*, pages 80–92, 2017.

[30] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. Event trend aggregation under rich event matching semantics. In *SIGMOD*, pages 555–572, 2019.

[31] O. Poppe, A. Rozet, C. Lei, E. A. Rundensteiner, and D. Maier. Sharon: Shared online event sequence aggregation. In *ICDE*, pages 737–748, 2018.

[32] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.

[33] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016.

[34] A. Rozet, O. Poppe, C. Lei, and E. A. Rundensteiner. Muse: Multi-query event trend aggregation. In *CIKM*, page 2193–2196, 2020.

[35] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[36] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8(7):702–713, 2015.

[37] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006.

[38] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in CEP. In *SIGMOD*, pages 217–228, 2014.

[39] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. Multi-query optimization for complex event processing in SAP ESP. In *ICDE*, pages 1213–1224, 2017.

[40] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 431–442, 2004.