Gloria: Graph-based Sharing Optimizer for Multi-Event Trend Aggregation

Anonymous

September 15, 2021

Abstract

Complex event processing is widely deployed to evaluate large workloads of pattern queries for applications from healthcare, transportation, to fraud detection. High-performance aggregation queries with KLEENE patterns are crucial to capture the high-level threats, alerts, or vital notifications. This work focuses on the problem of effective shared processing of multi-event trend aggregation queries under tight real-time constraints. In particular, we answer two critical challenges: (1) how to efficiently identify sharing opportunities for a variety of complex query patterns composed of advanced operators from KLEENE to nesting, and (2) how to effectively leverage these sharing opportunities to truly benefit the execution of the given workload. To the best of our knowledge, state-of-the-art methods fail to address both challenges. Hence, here we propose GLORIA, a graph-based sharing optimization for multi-event trend aggregation. By mapping the sharing optimization problem into a graph path search problem and cleverly encoding execution costs into the GLORIA graph edge representation. We design a suite of pruning rules based on the optimal path properties in a GLORIAgraph. These rules are exploited during and after the graph construction phase, which allows our innovative path search algorithm identify the sharing plan with minimal execution costs efficiently. Our experimental study on three real-world data sets demonstrates that our pruning rules effectively reduce the search space and contribute 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 [**?**, **?**] 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* [**?**], can be arbitrarily long and expensive to compute [**?**, **?**, **?**]. It is thus becoming increasingly difficult to support high-performance event processing due to the rising number and complexity of event trend aggregation queries [**?**].

Multi-query optimization for event trend aggregation is a promising approach [**?**, **?**, **?**] 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.

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

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

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 [**?**, **?**, **?**, **?**] 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 [**?**, **?**], 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 [**?**, **?**]. On the other hand, *online aggregation methods* [**?**, **?**, **?**, **?**, **?**] 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 [**?**]. 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 [**?**, **?**, **?**, **?**] restrict or completely disallow KLEENE patterns supported in the event trend aggregation. For example, MCEP [**?**, **?**] only support the sharing of non-nested (flat) KLEENE patterns with the pattern composed of only one single event type, while SHARON [**?**] 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 [**?**].

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 [**?**] introduces the concept of a sharing conflict, which does not allow one sub-pattern to participate in multiple sharing query groups. Similar to MCEP [**?**], 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 unnecessarily missed. Table 1 summarizes exemplar approaches categorized by the three dimensions discussed above.

Approach	**Aggregation strategy**	**KLEENE pattern type**	**Sharing decision**
MCEP [**?**]	two-step	restricted	flexible
SHARON [**?**]	online	–	restricted
GRETA [**?**]		general	–
HAMLET [**?**]		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 [**?**, **?**].

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 4, we introduce the core GLORIA graph model with pruning principles. In Section 5, we extend our GLORIA optimizer to support sharing optimization for general KLEENE patterns. We present experiments, related work, and conclusion in Sections 7, 8, and 9, respectively.

2 Preliminaries

2.1 GLORIA Query Model

Definition 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 (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 1;*
- *WHERE clause: Predicates θ; (optional)*

- *GROUPBY clause: Grouping G; (optional)*
- *WITHIN/SLIDE clause: Window w.*

An event trend of query q is a result sequence of events $tr = (e_1, \ldots, e_k)$ 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.

Notation	Description
Q	A workload of queries
$start(q)$	Start event type of q
$end(q)$	End event type of q
$pe(e, q)$	Predecessor events of e w.r.t q
$pe(E, q)$	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.

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.

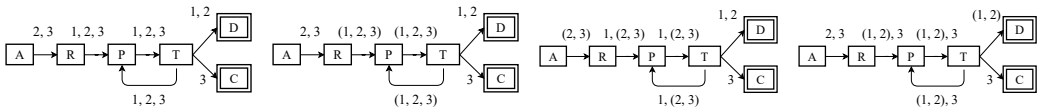
Definition 3 (Shareable Patterns)**.** *Given a pattern P and a workload Q of event trend aggregation queries, if P appears in the pattern clause of at least two queries, this pattern P is shareable.*

Example 1. *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[**?**]. 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 [**?**, **?**, **?**, **?**], 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 [**?**].



(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. *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 4 (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 5 (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 a workload Q and an event stream, the multi-event trend aggregation sharing problem Π is to find an optimal workload sharing plan that executes Q with minimal average latency per query compared to all other plans.

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 [**?**]:

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

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 GLORIA framework. The GLORIA optimizer takes as input a workload of queries and the stream statistics. The template constructor encodes the given workload into a GLORIA template that reveals the sharing opportunities (Section 2.2). By analyzing the GLORIA template, the graph constructor constructs a GLORIA graph to capture a variety of sharing opportunities from the GLORIA template (Section 4.1 and Section 4.2). This way, the workload sharing plan search space is transformed into the GLORIA graph, in which an optimal workload sharing plan corresponds to a path with the minimum weight in the graph. To reduce the

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

size of the search space, a set of pruning rules based on the cost model are utilized by the graph constructor (Sections 4.3 and 5). Finally, the sharing plan generator utilizes the path searching algorithm to find the optimal workload sharing plan in the GLORIA graph (Sections 4.4 and 5). During the path searching, the pruning rules are leveraged by the sharing plan generator to further reduce the search time to linear in the size of the graph.

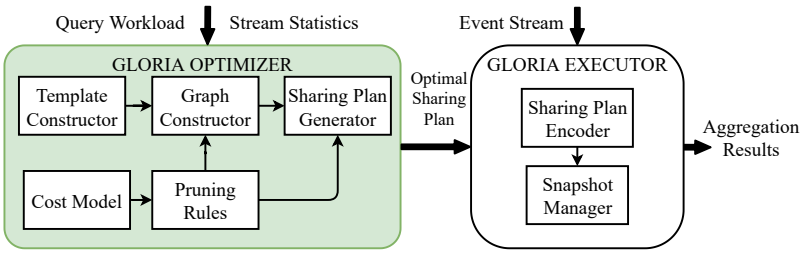


Figure 3: GLORIA framework.

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.

3.2 GLORIA Executor

In this paper, GLORIA executor adopts the state-of-art online aggregation methods [**?**, **?**] 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. *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 and b_2 arrive, their intermediate aggregate are obtained by propagation from their predecessors c_0 and a_1, respectively. Similarly, when a_3 arrives, its predecessors b_2 and c_0 propagate their intermediate aggregate to a_3. The value of a_3's aggregate becomes 2. And finally, when d_5 arrives (in orange color indicating it is the end event of q_1), the intermediate aggregates of b_2 and b_4 are propagated to d_5 and summed up as the final count $fcount = 4$ for q_1.*

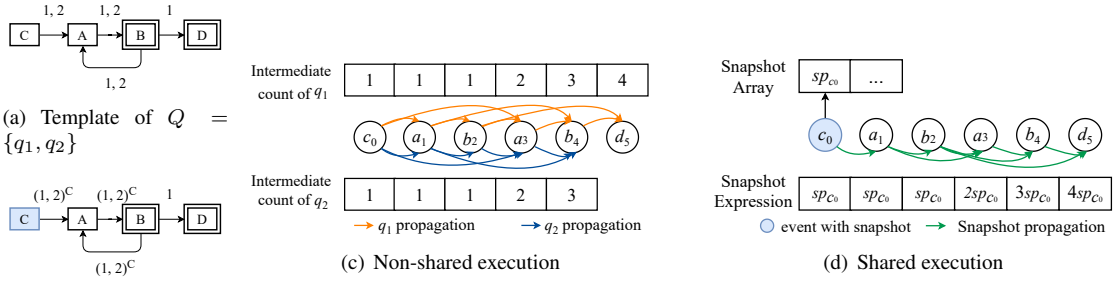
The execution cost of q_1 and q_2 lies in the propagation of intermediate aggregates from the predecessors to the newly arrived matched events. For example, $cost(a_1, q_1)$ = $cost(a_1, q_2)$ = 1 and $cost(a_3, q_1)$ = $cost(a_3, q_2)$ = 2. As shown in Figure 4(c), the non-shared execution cost of all 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

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

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* [**?**] 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



(a) Template of $Q = \{q_1, q_2\}$

(b) A sharing plan for Q

(c) Non-shared execution

(d) Shared execution

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

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 4. *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 evaluation. 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:

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

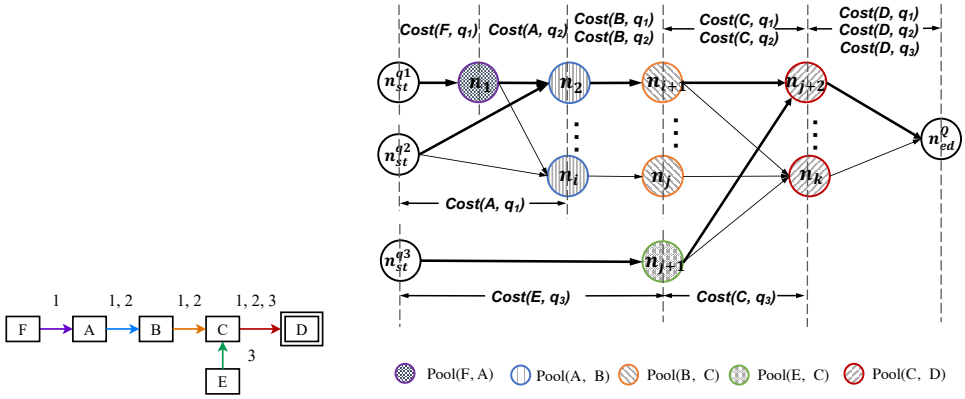
4 GLORIA Optimizer

In this section, we introduce the foundation of the GLORIA optimizer. Given a GLORIA template for SEQ or KLEENE patterns, our optimizer constructs a GLORIA graph to capture the search space of workload sharing plans. During graph construction, we propose three node generation principles to limit the number of nodes created in the graph. We further present two universal pruning rules of nodes and edges, and one KLEENE-specific path pruning rule to reduce the size of the GLORIA graph. Then with the pruned GLORIA graph, our path searching algorithm is applied to find the optimized workload sharing plan on the GLORIA graph.

4.1 GLORIA Graph Model

Given a GLORIA template of Q, a workload sharing plan decides which queries to be shared on each transition in the template. We use GLORIA graph to capture the search space of workload sharing plans.

Definition 6 (GLORIA Graph). *A GLORIA Graph is a weighted directed graph with a set of nodes and a set of directed edges. A node n_m represent either a transition sharing plan $tran(E_i, E_j)$, in which E_i and E_j are two adjacent event types, a start node n^q_{st} or an end node n^q_{ed} in GLORIA template. A pool $Pool(E_i, E_j)$ consists of a set of nodes corresponding to all candidate sharing plans for E_i and E_j. A directed weighted edge $edge(n_k, n_m)$ connects two nodes n_k and n_m, if they belong to two consecutive pools. Let $\bar{Q}$ be the common queries in n_k (i.e., $tran(E_i, E_j)$) and n_m (i.e., $tran(E_j, E_h)$), the weight of the edge $edge(n_k, n_m)$ represents the execution cost of E_j for $\bar{Q}$ ($Cost(E_j, \bar{Q})$), when applying the transition sharing plans n_k and n_m.*



(a) Template (b) GLORIA graph

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

Example 5. *Figure 5 depicts the GLORIA 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 $n_m \in Pool(B, C)$.*

Definition 7 (GLORIA Path). *Given a GLORIA graph, a GLORIA 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(b) illustrate an example path, namely one workload sharing plan for Q. The weight of a path captures the total execution cost of the workload sharing plan.

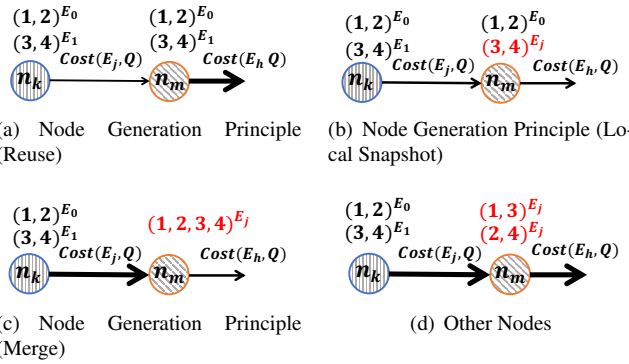
[me: Lemma 1. A path's weight corresponds to the execution cost of the workload sharing plan it represents. Lemma 2. The paths in the graph cover all possible workload sharing plans. Lemma 3. The minimal weighted path corresponds to optimal workload sharing plan. Proofs of 1,2 refer to technical report. Lemma 3 gives the current short proof.]

Lemma 4.1. *The optimal path $\bar{P}$ with the minimum weight, denoted as $\bar{P}.w$, in a GLORIA graph corresponds to the optimal workload sharing plan with minimum execution cost.*

Proof. We first prove that a GLORIA path P captures the execution cost of the corresponding workload sharing plan. Given a GLORIA graph of workload Q and a GLORIA path P, and one node $n_m \in pool(E_j, E_h)$, the weight of each incoming edge of n_k covers the execution cost of E_j for each $E_i \in pe(E_j, Q)$. Therefore, all incoming edges covers the execution cost of E_j for all propagations from $E_i \in pe(E_j, Q)$. By the definition of path, P visits every pool in the graph that every node covers the execution cost of an event type. Therefore, given a GLORIA path P, its weight $P.w$ captures the execution cost of the corresponding workload sharing plan. Based on that, the optimal path $\bar{P}$ with minimum weight corresponds to the optimal workload sharing plan. □

4.2 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.



(a) Node Generation Principle (Reuse)

(b) Node Generation Principle (Local Snapshot)

(c) Node Generation Principle (Merge)

(d) Other Nodes

Figure 6: Node generation

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 6. *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 4.2. *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.*

[me: expand this proof]

Proof. In case 1, compared with keep the *Q-set*s of n_k, the breaking of *Q-set*s involves expensive expression evaluation for E_j, which increases the weight of the incoming edge. Also, by increasing number of *Q-set*s, fewer queries are shared together which in consequence increases the number of propagations for E_h, which increases the weight of the outgoing edge. In case 2, shuffling the *Q-set*s also introduces evaluation. To keep the sharing status of the shuffled new *Q-set*s, each *Q-set* needs to use E_j as the snapshots. Compared with merging these *Q-set*s into a bigger *Q-set* with snapshots of E_j, multiple *Q-set*s with snapshots of E_j have more propagations, which leads to heavier outgoing edge. □

Lemma 4.2 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 7. *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:*

$$\begin{aligned} edge(n_k, n_m).w &= Cost(E_j, Q) \\ &= Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \end{aligned} \tag{5}$$

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 8. *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:*

$$\begin{aligned} edge(n_k, n_m).w = Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \\ + Cost_{eval}(E_j, \bar{Q}_2) \end{aligned} \tag{6}$$

Compared with Equation 5, 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 9. *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:*

$$\begin{aligned} edge(n_k, n_m).w = Cost_{share}(E_j, \bar{Q}_1) + Cost_{share}(E_j, \bar{Q}_2) \\ + Cost_{eval}(E_j, \bar{Q}_1) + Cost_{eval}(E_j, \bar{Q}_2) \end{aligned} \tag{7}$$

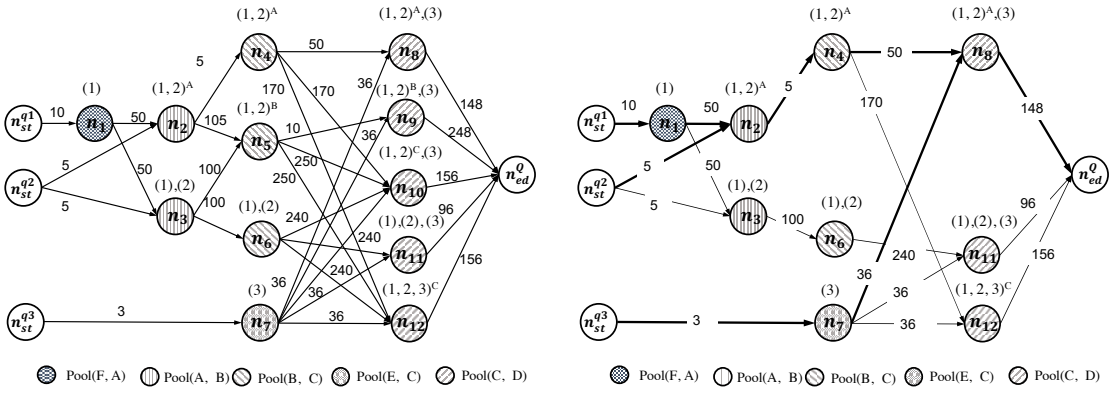
Compared with Equation 5 and 6, the $edge(n_k, n_m)$ has the heaviest weight for merging, but n_m may have the minimum weight of outgoing edge compared with the other two cases.

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. Due to space limitation, we elaborate this process in our technical report [**?**].

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 10. *Figure 7(a) shows the* GLORIA *graph of template in Figure 5(a). 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}.



(a) Full Gloria Graph

(b) Pruned Gloria Graph With Optimal Path

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

4.3 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:

$$n_m.d_s = \sum_{E_i} min\{n_k.d_s + edge(n_k, n_m).w\}, \quad n_k \in Pool(E_i, E_j), E_i \in pe(E_j, Q) \tag{8}$$

Edge Pruning Principle. Given a node $n_m \in Pool(E_j, E_h)$, for each preceding $Pool(E_i, E_j), E_i \in pe(E_j, Q)$, if there are multiple edges from nodes of $Pool(E_i, E_j)$ to n_m, only $edge(n_k, n_m)$ in Equation 8 is kept. Other edges to n_m can be safely pruned.

Example 11. *Consider $Pool(B, C)$ in Figure 7(a) as the targeting pool. Table 3 lists d_s of 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 8, $n_2.d_s + edge(n_2, n_5).w = 170$ 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 4.3. *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:

$$\begin{aligned} \bar{P}.w &= P_1.w + P_2.w \\ &= n_0.d_s + edge(n_0, n).w + P_2.w \end{aligned} \tag{9}$$

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 8 (*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 $n_k.d_e < n_m.d_e$. The node n_m can be safely pruned compared with n_k, if they meet the following conditions:

1. $n_k.d_e < n_m.d_e$
2. $n_k.d_s < n_m.d_s$

Example 12. *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 $|A| < |B|$. Since n_4 is sharing the same queries with fewer snapshots, according to our cost model in Equation 3 and 4, no matter these snapshots will be reused or evaluated, the execution cost of sharing Q-set $(1, 2)$ with snapshots of A is lower than sharing Q-set $(1, 2)$ with snapshots of B. In this case, we can tell that $n_4.d_e < n_5.d_e$, even though the exact cost is still unknown. Also, according to Table 3, $n_4.d_s < n_5.d_s$. Thus, n_5 can be pruned compared with n_4.*

[me: expand this proof]

Lemma 4.4. *The node pruning principle does not exclude any optimal path on* GLORIA *graph.*

Lemma 4.4 can be proven by applying the cost model. Due to the limited space, we put proof of Lemma 4.4 in our technical report [**?**].

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 4.4, we apply our path searching algorithm on such pruned graph to find the optimal path.

4.4 Path Searching Algorithm

Given a pruned GLORIA graph G, Algo 1 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. Due to the limited space, we put these utility algorithms in our technical report [**?**].

Algorithm 1 GLORIAPATHSEARCH(G)

Input: GLORIA graph G
Output: The optimal path $\bar{P}$

```
1: P̄ ← An empty edge list, minPathWeight ← +∞
2: if G.getEndNodes().size = 1 then
3:     // Case 1: one end node
4:     endNode ← G.getEndNodes().get(0)
5:     EDGEPRUNE(endNode)
6:     P̄ ← GETPATHS().get(0)
7: else
8:     onePath ← true
9:     for each endNode ∈ G.getEndNodes() do
10:        if endNode.getIncomingeEdges().size > 1 then
11:            onePath ← false
12:    if onePath = true then
13:        // Case 2: multiple end nodes with only one path
14:        P̄ ← GETPATHS().get(0)
15:    else
16:        // Case 3: multiple end nodes with multiple paths
17:        REVERSEEDGES(G)
18:        for each n ∈ G do
19:            EDGEPRUNE(n)
20:        for each path ∈ GETPATHS() do
21:            if path.w < minPathWeight then
22:                P̄ ← path
23:                minPathWeight ← P̄.w
24:            else Continue
25: return P̄
```

[me: add utility algos]

The main algorithm GLORIAPATHSEARCH 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 $\bar{P}$.

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 GLORIAPATHSEARCH is $O(2N+T)$.

5 GLORIA Optimizer for Kleene Plus

Based on the foundation of GLORIA optimizer in Section 4, 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 8(a), 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.

Assumptions. To focus on core concepts, we assume that (1) a query in the workload contains the whole KLEENE sub-pattern and (2) one event type only appear once in a query. We refer to our technical report [**?**] for a generalized discussion.

5.1 Flat Kleene Patterns

According to our GLORIA graph model, two nodes from consecutive pools are connected. Therefore, there are edges from nodes in $Pool(A,B)$ to nodes in $Pool(B,A)$ and vice versa in the template in Figure 8(a). These edges create a cycle in the KLEENE sub-graph. This path is beneficial when all nodes in it have the same sharing plan.

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 $Pool(E_i,E_{i+1})(0\le i<k)$ 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 $E_i(0\le i\le k)$ for snapshots. Otherwise, all nodes are not shared.

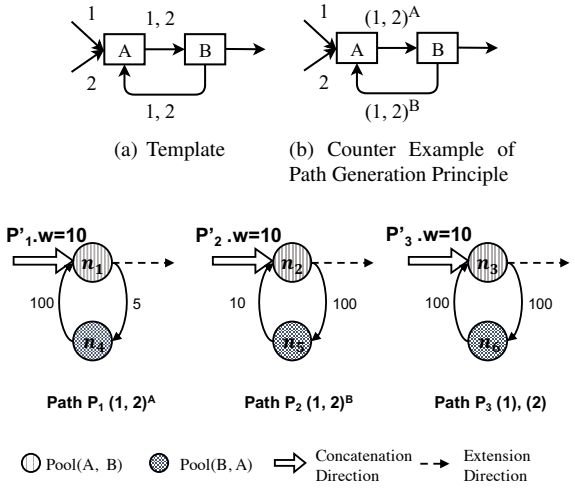
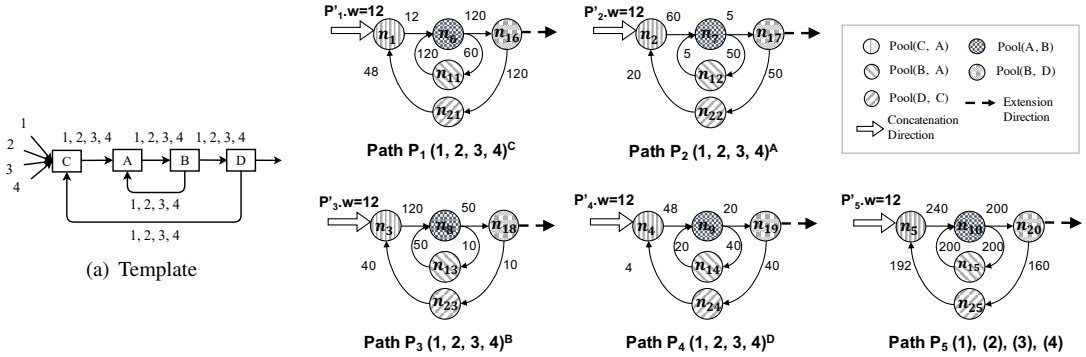


Figure 8: Gloria sub-graph of flat Kleene pattern for $Q=\{q_1,q_2\}$
Stream statistics: $|A|=5$, $|B|=10$

Example 13. *Consider Figure 8(b). The sharing plan on $tran(A,B)$ shares $(1,2)$ with snapshots of A and the sharing plan on $tran(B,A)$ shares $(1,2)$ with snapshots of B, where $|A|<|B|$. Due to Kleene cycle, creating local snapshots of B on $tran(B,A)$ introduces evaluation cost for both $tran(A,B)$ and $tran(B,A)$. Since*

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) 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: $|A| = 5$, $|B| = 10$, $|C| = 12$, $|D| = 4$

$|B| < |A|$, *replacing snapshots of* A *by snapshots of* B *brings no sharing benefit. The execution cost of this example is larger than sharing* $(1,2)$ *with snapshots of* A *for both transitions. Figure 8(c) shows all generated paths following Path generation principle. Specifically,* 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 hollow arrows* P'_1, P'_2, P'_3 *indicate the concatenation direction to the* KLEENE *sub-graph for each path, together with the respective weight. The dashed arrow indicates the extension direction of the* KLEENE *sub-graph for each path. By adding the weights of edges in the path, we have* $P_1.w = 105$*,* $P_2.w = 110$ *and* $P_1.w = 200$*.*

Lemma 5.1. *Path generation principle does not discard optimal path in* KLEENE *sub-graph.*

Proof. Since we optimize the KLEENE sub-pattern independently, the decision for *Q-set*s 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 4.2, 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. □

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:

$$n_k.d_s = P'_i.w + P_i.w \tag{10}$$

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 14. *Continuing Example 13, assume* P'_1 *to* P'_3 *have the same weight 10. Table 4 lists the source nodes* d_s *of* n_1 *to* n_3 *in* $Pool(A,B)$*. According to Node pruning principle,* n_2 *is pruned compared with* n_1*, together with the whole path* P_2*.*

5.2 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 SEQ$(C,$ 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 5.2. *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 5.1, 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 5.2, 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*.

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

Example 15. *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}.*

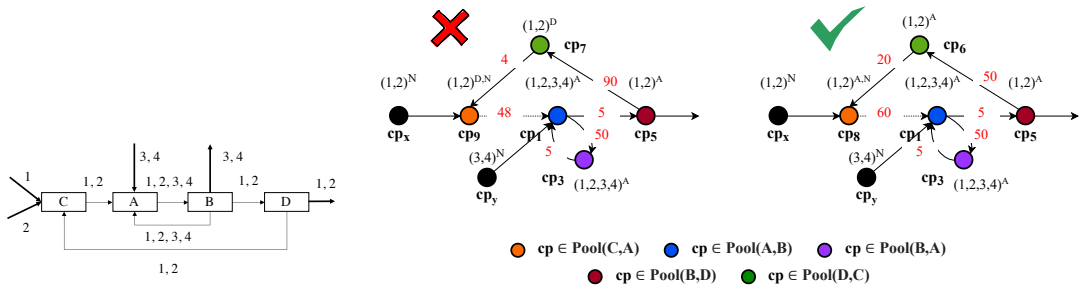
Path Search. Given that paths in the KLEENE sub-graph are cycles, they require minor modification to only GETPATHS without affecting the main algorithm GLORIAPATHSEARCH (Algorithm 1). Specifically, Case 1 in GLORIAPATHSEARCH remains the same since EDGEPRUNE only prunes the multiple incoming edges for 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, GLORIAPATHSEARCH 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.

5.3 Overlapping KLEENE sub-patterns

Layer Shared Nested Cycle. A layer shared nested cycle is a cycle generated in a template when a nested kleene pattern is sharing with its sub kleene patterns. Figure 10(a) illustrates a partial template of such cycle. In a workload $Q = \{q_1, q_2, q_3, q_4\}$, q_1, q_2 have nested pattern $(C, (A, B)+, D)+$ but q_3, q_4 only have the inner sub kleene pattern $(A, B)+$. Such hierarchical sharing should still be supported. But it brings the following challenges.

• *Exponential Number of Cycle Sharing Plans.* Lemma 4 guarantees the linear number of cycle sharing plans. However, the different query sets on each edge brought by the hierarchical sharing violate the condition of Lemma **??**, which causes exponential number of cycle sharing plans. We prove Lemma **??** by using Transition Rules for single S-set, however, here we prove that the transition rules doesn't apply for multiple S-sets. According to Transition Rule, given a S-set $(1, 2)^X$, it cannot transit to $(1, 2)^Y$ when $|X| < |Y|$. However, this rule doesn't apply when it comes to multiple S-sets. For instance, let a node has two S-sets $(1, 2)^{X_1}$ and $(3, 4)^{X_2}$, this node may transit to $(1, 2, 3, 4)^Y$ when $|X_1| < |Y| < |X_2|$. For S-set $(1, 2)$, such transition is non-beneficial since new Y snapshots is more than the old X_1 snapshots, together with the extra evaluation cost. But for S-set $(3, 4)$ such transition may be beneficial since $Y < |X_2|$. Moreover, the merging of two S-sets introduces extra benefit, since more queries are shared together, less re-computation for the predecessor reaching.



(a) Layer Shared Nested Cycle

(b) Cycle paths of Layer Shared Nested Cycle

Figure 10: Layer Nested Cycle Graph

Together, it's possible that a node with multiple S-sets $(1,2)^{X_1}(3,4)^{X_2}$ could transit to $(1,2)^{Y_1}(3,4)^{Y_2}$ where $|Y_1| < |X_1|, |Y_2| < |X_2|$, or $(1,2,3,4)^Y$ where event $|Y| > max\{|X_1|, |X_2|\}$. For example, in Figure 10(a), a $cp = (1,2)^D \in Pool(D,C)$ could only transit to $cp = (1,2)^D \in Pool(C,A)$ due to transition rules, however, when it comes to $Pool(A,B)$, $cp = (1,2)^D \in Pool(C,A)$ could transit to $(1,2,3,4)^A \in Pool(A,B)$ even if $|A| > |D|$. (should cut down the details.)

Due to the possibilities of merging, number of transitions could be exponential. For a multiple layered nested cycle, let the outer cycle be the 1-st layer. Assume each layer introduces a new S-set, for the n-th layer, we have n S-sets. Among all n S-sets, arbitrary S-sets could be merged together. The total number of merging possibilities is the n-th Bell Number which is $O(e^{e^n})$.

- *Multiple Optimization Directions*. Not only multiple event types with incoming edges, q_3, q_4 also bring outgoing edges to event type B. In the whole nested cycle, both event types B and D are holding outgoing edges and each event type corresponds to an optimization direction. These multiple event types breaks Property 2 of Lemma **??** which means even the cycle graph is constructed, by the current construction and pruning process, it cannot guarantee global optimal for the whole Gloria Graph. This limitation can be explained by induction. Due to B, D holding outgoing edges, both $cp \in Pool(A,B)$ and $cp \in Pool(B,D)$ are output nodes, during pruning, each pool will prune out the local non-beneficial nodes which may cause conflicts. Pruning a non-beneficial node in one pool may cause a beneficial node in the other pool gets pruned. Also, since the pruning is local, global optimality cannot be guaranteed without considering all the possibilities. With more layers in the nested cycle, more optimization directions we have, more complicated the global optimization becomes.

Therefore, instead of aiming for global optimal, we switch our optimization goal to optimize as many edges as possible. One property about the layer shared nested cycle is that the edge query set of inner cycle is always the super set of the edge query set of the outer cycles. If the whole query set is shared with some snapshot, this plan can be reused by all the edges in the outer cycles. Inspired by this property, We propose ***Spiral Optimization.*** Instead of optimizing outer to inner, spiral optimization firstly constructs and prunes a cycle graph for the innermost cycle. After that, along the following SEQ and feedback edges, it constructs the respective pools. Finally, it circles back along the feedback edge, constructing left SEQ edges from outer cycles to the innermost one. how many cycle sharing plans we have. how good the optimization is. local optimal for the innermost cycle

Example 16. *Figure 10(b) shows the two cycle paths of Figure 10(a) found by the spiral optimization. Instead of starting with $Pool(D,C)$, the spiral optimization starts with $Pool(B,A)$ that $cp_1 = (1,2,3,4)^A$ and $cp_2 = (1,2,3,4)^B$. Therefore, inner cycle $(A,B)+$ has two cycle paths $Path_c^1 = (cp_1, cp_3, cp1)$ using snapshot A and $Path_c^2 = (cp_2, cp_4, cp_2)$ using snapshot B. After pruning, all nodes and transitions in $Path_{cycle}^2$ is pruned.*

With the cycle graph of $(A,B)+$, we optimize the right edges (B,D) and (D,C) of the inner cycle. For $Pool(B,D)$, according to transition rule 1, only $cp_5 = (1,2)^A$ is generated from cp_1. For $Pool(D,C)$, both $cp_6 = (1,2)^A$ and $cp_7 = (1,2)^D$ can be generated from cp_5.

After optimizing the inner cycle and all following SEQ and Kleene edges, we optimize the left SEQ edges from outer cycle to inner cycle. In this case, for $Pool(C,A)$, $cp_8 = (1,2)^A$ is generated from cp_6 and cp_x as well as $cp_9 = (1,2)^D$ is generated from cp_7 and cp_x.

*The graph generates two cycle plans as shown in Figure **??**, Table **??** lists their weights. By Equation **??**, eCost of output nodes cp_1 and cp_5 are computed as in Table **??**. During pruning, $Path_c^1$ is pruned. elaborate.*

6 Discussion

In this section, we discussion how to extend GLORIA with relaxed assumptions.

Aggregations. For aggregation functions like SUM($E.\ attr$), MIN($E.attr$), MAX($E.attr$), the aggregate is updated by summing, or comparing with the attribute values of a new event of E for all partial event trends it extends. For AVG($E.attr$), we maintain SUM and COUNT as the intermediate values and output AVG when output the final value for queries. Therefore, queries with SUM($E.attr$), AVG($E.attr$) and COUNT($*$) can be shared together.

Predicates. Given a workload, GLORIA analyzes the compatibility of predicates. Our proposed method applies for compatible predicates. If the shareable queries have incompatible predicates, no sharing opportunities can be leveraged, GLORIA still can apply the non-sharing execution.

Windows. For different windows in a workload, GLORIA partitions the window into ***panes*** that are shareable over overlapping windows [**?**, **?**, **?**, **?**], whose size is the greatest common divisor of the window sizes and window slides.

7 EXPERIMENTAL EVALUATION

7.1 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 [**?**]. 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)* [**?**] 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)* [**?**] 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)* [**?**] 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.

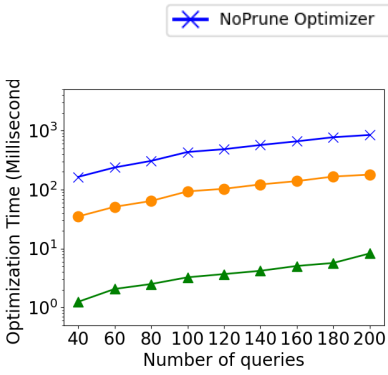
• *Greedy Optimizer (Greedy).* For each pool, the *Greedy* optimizer only considers the transition sharing plan with the minimum incoming edge weight (Section 4.2), thus potentially missing sharing benefits. Specifically, with respect to node generation, it either applies *Node generation principle (Reuse)* 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 4.3). 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.

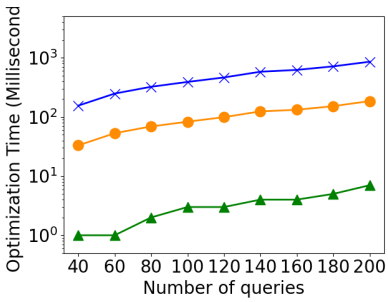
We also evaluate the execution of the optimized sharing plans generated by GLORIA optimizer. We first compare it with the execution of the plan generated by the *Greedy* optimizer, and then compare it with GRETA[**?**], a state-of-the-art method supporting online aggregation over nested KLEENE patterns.

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.

7.2 GLORIA Optimization Evaluation

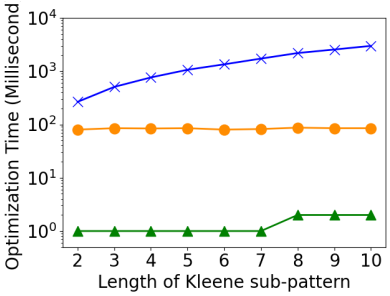


(b) Bus data set

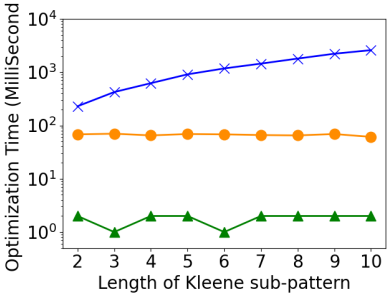


(c) Taxi data set

Figure 11: Varying # queries (SEQ)



(a) Bus data set



(b) Taxi data set

Figure 12: Varying length of KLEENE sub-patterns (KLEENE)

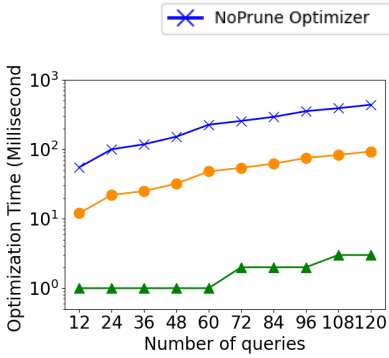
SEQ Workload. To evaluate the effectiveness of pruning rules on SEQ patterns, we measure the optimization time of three optimizers on *Bus* and *Taxi* data sets in Figure 11 while varying the number of queries in the SEQ workload from 40 to 200. On both data sets, GLORIA optimizer consistently outperforms 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 *[me: optimal]* 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 4.3). 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 *[me: the optimality guarantee]*.

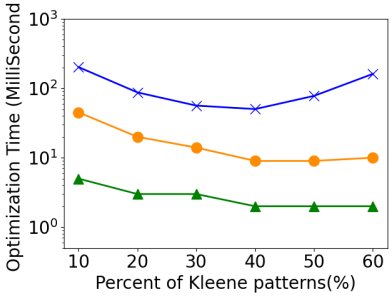
KLEENE Workload. To evaluate the effectiveness of pruning rules on KLEENE patterns, we compare three optimizers on the KLEENE workload with 100 queries in Figure 12 while varying the length of KLEENE sub-patterns from 2 to 10. GLORIA consistently outperforms *Greedy* optimizer and *NoPrune* optimizer. 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 5.1. 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 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.

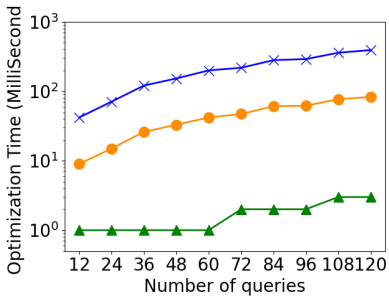
Mixed Workload. Figure 13 compares the three optimizers on mixed workloads with a SEQ-to-KLEENE ratio as 6:1 with varying number of queries. Again, GLORIA optimizer constantly outperforms *NoPrune* optimizer by 5-fold and is slower than the *Greedy* optimizer by 1 order of magnitude. In Figure 14, we 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 5), the optimization time of GLORIA optimizer decreases as the percent of KLEENE queries increases. Due to the limited space, we put the result of memory consumption into our technical report[**?**]. Due to the pruning rules, GLORIA optimizer only consumes 25% memory of the *NoPrune* optimizer on both data sets.



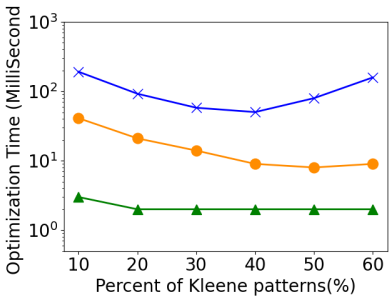
(b) Bus data set



(a) Bus data set



(c) Taxi data set



(b) Taxi data set

Figure 13: Varying # queries (time)

Figure 14: Varying percent of KLEENE queries (time)

7.3 GLORIA Runtime Evaluation

To evaluate the quality of the sharing plan produced by GLORIA, we evaluate several mixed workloads over all three data sets, shown in Figure 15. 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.

We measure throughput in Figures 15(c), 15(e) and 15(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 15(d) and 15(f)) or performs similarly (Figure 15(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.

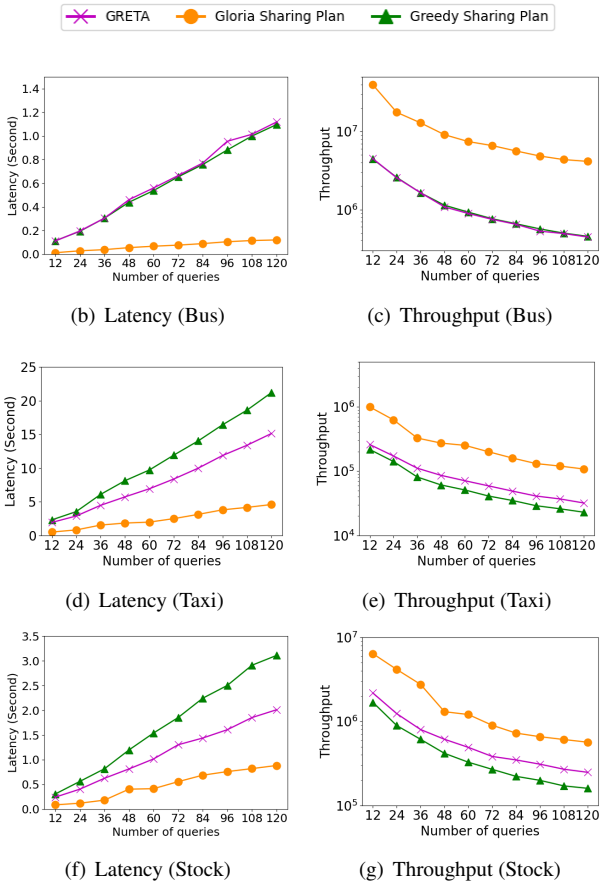
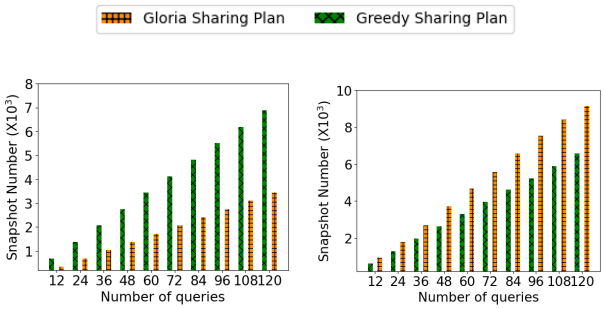


Figure 15: Execution with different sharing plans (Mixed)

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 16 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 16(b)) and more snapshots than *Greedy* in Figure 16(c).



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

Figure 16: # 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 16(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 16(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.

8 Related Work

Complex Event Processing Systems. CEP have gained popularity in the recent years [**?**, **?**, **?**, **?**]. Some approaches use a Finite State Automaton (FSA) as an execution framework for pattern matching [**?**, **?**, **?**, **?**]. Others employ tree-based models [**?**]. Some approaches study lazy match detection [**?**], compact event graph encoding [**?**], and join plan generation [**?**]. We refer to the recent survey [**?**] 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 [**?**, **?**, **?**, **?**]. A-Seq [**?**] introduces online aggregation of event sequences, i.e., sequence aggregation without sequence construction. GRETA [**?**] extends A-Seq by Kleene closure. Cogra [**?**] 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 [**?**], pattern sharing techniques for CEP have attracted considerable attention. RUMOR [**?**] defines a set of rules for merging queries in NFA-based RDBMS and stream processing systems. E-Cube [**?**] inserts sequence queries into a hierarchy based on concept and pattern refinement relations. SPASS [**?**] estimates the benefit of sharing for event sequence construction using intra-query and inter-query event correlations. MOTTO [**?**] applies merge, decomposition, and operator transformation techniques to re-write pattern matching queries. Kolchinsky et al. [**?**] combine sharing and pattern reordering optimizations for both NFA-based and tree-based query plans. Most recently, HAMLET [**?**] 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 [**?**, **?**, **?**] 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 [**?**], Muse [**?**], and HAMLET [**?**] are the only solutions that support shared online aggregation. However, SHARON does not support Kleene closure, MCEP and COGRA [**?**, **?**] 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.

9 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