Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation

Lei Ma[1], Chuan Lei[2], Olga Poppe[3], Elke A. Rundensteiner[1]
[1]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
[2]Instacart, 50 Beale Street Suite 600 San Francisco, CA 94105
[3] Microsoft Gray Systems Lab, One Microsoft Way, Redmond, WA 98052
lma5@wpi.edu,chuan.lei@instacart.com,olpoppe@microsoft.com,rundenst@wpi.edu

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.

CCS CONCEPTS

• **Information systems → Data stream mining**; • **Computing methodologies → Optimization algorithms**.

KEYWORDS

Complex event processing; query optimization; computation sharing; incremental aggregation; event trend

ACM Reference Format:
Lei Ma, Chuan Lei, Olga Poppe and Elke A. Rundensteiner. 2022. Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526145

1 INTRODUCTION

Complex Event Processing has shown great promise in retrieving insights over high-velocity event streams in near real time for applications ranging from transportation to public health. These applications often rely on aggregation queries composed of *Kleene* [2, 7] and other operators that express dependencies among event types to retrieve summarized information of interest. Unlike fixed length event sequences, the sequences matched by *Kleene* patterns, called *event trends* [25], can be arbitrarily long and expensive to compute [21, 26, 27]. Thus, it is difficult to guarantee real time response of large workloads of such queries [3].

Multi-query optimization for event trend aggregation is a promising approach [21, 26, 29] to reduce the query execution costs by sharing computations among queries in a workload. Given an event trend aggregation workload, such optimization technique must (1) identify sharing opportunities in nested Kleene patterns and (2) decide how to leverage these sharing opportunities to truly benefit the execution of the given workload.

Motivation Example. Figure 1 shows an example workload from a food delivery application. An event type corresponds to an action made by the customer or the delivery driver, e.g., *AppOrder* or *WebOrder*, *Request* or *Travel*. Each event in the stream is a tuple composed of a customer/driver identifier, an action, a district, and a timestamp. The queries q_1, q_2 and q_3 differ by the patterns shown at the bottom, while all other query clauses are the same. There can be any number of consecutive (*Pickup*, *Travel*) sequences expressed by *Kleene*. Query q_1 focuses on the orders that are requested and delivered. Query q_2 detects the orders that are placed on the app end and finished by delivery. Query q_3 tracks the orders are cancelled by the customer. All three queries contain the *Kleene* sub-pattern *Request*, (*Pickup*, *Travel*)+, which could be shared among them. However, this is not the only sharing opportunity. In fact, q_1 and q_2 could also share the longer sub-pattern (*Request*, (*Pickup*, *Travel*)+, *Delivery*). Alternatively, q_2 and q_3 can share a different longer SEQ sub-pattern (*AppOrder*, *Request*, (*Pickup*, *Travel*)+). To share such queries with both *Kleene* and SEQ sub-patterns with different sharing opportunities is not a trivial task.

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

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

Figure 1: Event Trend Aggregation Workload

State-of-the-Art Approaches. Early works [18, 21, 25, 36] use a *two-step approach* that first constructs all event trends and then aggregates them. While the event trend construction step may be shared by several queries [18, 21], these two-step methods suffer from an exponential time complexity in the number of matched events [25, 36]. Instead, *online approaches* [26–29, 32] hold promise by pushing aggregation into the pattern matching phase without having to first construct event trends. This reduces the computation complexity from exponential to quadratic [27].

However, the state-of-the-art online aggregation methods suffer from two critical limitations. First, *Kleene* patterns can contain nested SEQ and *Kleene* operators. Hence, most existing methods [21, 26, 28, 29] restrict or completely disallow *Kleene* patterns in the event trend aggregation. For example, MCEP[21] only supports the sharing of non-nested (flat) *Kleene* patterns, while SHARON[29] supports sharing fixed-length SEQ patterns only, i.e., no *Kleene* patterns. Such restrictions on the *Kleene* patterns greatly reduce the applicability of existing sharing methods.

Approach	**Aggregation strategy**	***Kleene* pattern type**	**Sharing decision**
MCEP [21]	two-step	restricted	flexible
SHARON [29]	online	–	restricted
GRETA [27]		general	–
HAMLET [26]		restricted	restricted
GLORIA (our)		**general**	**flexible**

Table 1: Event trend aggregation approaches.

Second, the state-of-the-art methods make strict assumptions about sharing decisions. For example, SHARON [29] introduces the concept of sharing conflict, which does not allow one sub-pattern to participate in multiple sharing query groups. Similar to MCEP[21], HAMLET[26] only considers sharing opportunities among flat *Kleene* sub-patterns containing a single event type. These assumptions often result in sub-optimal sharing plans, since many sharing opportunities are missed. Table 1 summarizes exemplar approaches categorized by the three dimensions discussed above.

Challenges. We aim to address the following open challenges.

Query complexity. In real-world applications, a query workload often consists of nested *Kleene* operators applied to a sequence of event types. A sharing optimizer needs to discover sharing opportunities among those complex *Kleene* queries, despite the exponential complexity of the problem [25, 36].

Sharing complexity. Sharing opportunities must be fully exploited without rigid constraints. Moreover, a cost model is needed to accurately capture the execution cost of different sharing opportunities. A sharing optimizer can only harvest the maximal sharing benefit when it is able to identify truly beneficial sharing opportunities among complex event trend aggregation queries.

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 optimality.

Proposed Solution. Given a workload of event trend aggregation queries composed of nested SEQ and *Kleene* operators, the GLORIA optimizer generates a fine-grained sharing plan that decides which queries should share which sub-patterns depending on the event stream characteristics. Specifically, by mapping the sharing optimization problem into a graph path search problem, the GLORIA optimizer generates a GLORIA graph that captures the sharing plan search space. The GLORIA optimizer leverages a set of universal pruning rules for SEQ and *Kleene* sub-patterns with additional *Kleene* -specific rules to construct a compact GLORIA graph. Lastly, the path search algorithm further prunes the graph and finds the optimal path in linear time in the size of the graph.

Contributions. Our main contributions are the following.

1. We present a novel approach for optimizing the sharing plan for a workload of event trend aggregation queries with complex *Kleene* patterns. To the best of our knowledge, GLORIA is the first sharing optimizer that offers flexible sharing decisions on nested *Kleene* patterns using effective online aggregation.

2. We introduce GLORIA graph to model a variety of sharing opportunities in a diverse event trend aggregation workload. This allows us to transform the workload sharing problem into a path search problem. A cost model and effective pruning rules are also introduced to reduce the size of the GLORIA graph.

3. We design a path search algorithm to find an event trend aggregation sharing plan from the GLORIA graph. For a given complex query workload, our optimizer has a linear-time complexity in the number of nodes and edges in the GLORIA graph, while still delivering optimality guarantees in most cases.

4. 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 up to 10x performance improvement over the state-of-the-art approaches.

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.

Notations	**Descriptions**
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
$edge(n_k, n_m)$	Edge between n_k and n_m
$n_k.d_s$	Minimum distance from all start nodes to n_k
$n_k.d_e$	Minimum distance from n_k to all end nodes
$P.w$	Weight of a path P in GLORIA graph

Table 2: Table of notations.

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.

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. Given a 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 the start and end event types in q, respectively. Table 2 summarizes the notations.

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 [16], i.e., that are associative and commutative. Specifically, COUNT(*) returns the number of matched event trends, MIN/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 these events. For simplicity, we use COUNT(*) as the default aggregation function in this paper. We discuss sharing of other aggregation functions in the discussion section (Sec. 7).

2.2 Event Trend Aggregation Sharing Problem

To facilitate the analysis of sharing opportunities, we represent a query workload as a Finite-State-Automaton [12, 14, 35, 36], called *Gloria Template.* Each node in the template represents an event type E 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 E_i 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 [27].

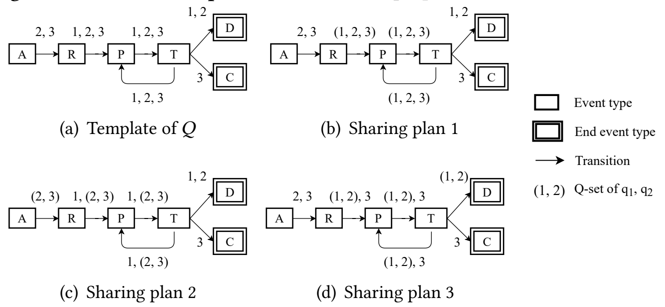


Figure 2: Gloria template and sharing plans of $Q = \{q_1, q_2, q_3\}$

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. Intuitively, a transitions[1] in the template can be shared by different queries. The set of queries shared together is called a ***Q-set***. The sharing plan 1 in Figure 2(b) shares transitions $tran(R, P)$, $tran(P, T)$ and $tran(T, P)$ for all three queries, denoted as $(1, 2, 3)$. Alternatively, the sharing plan 2 in Figure 2(c) shares transitions $tran(R, P)$, $tran(P, T)$ and $tran(T, P)$ between q_2 and q_3. Figure 2(d) shows that q_1 and q_2 share transitions $tran(R, P)$, $tran(P, T)$, $tran(T, P)$ and $tran(T, D)$.

This example raises three questions. (1) How can we effectively share *Kleene* sub-patterns? (2) Could consecutive transitions share different queries? (3) Overall, how can we determine the costs of alternate sharing plans and then find the optimal plan among them? Below we introduce the concepts of transition and workload sharing plan to specify the required information for the above questions.

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 modeled by this transition respectively.

Definition 2.7 (Workload Sharing Plan). Given a 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 event trend aggregation sharing problem is to find a workload sharing plan composed of pattern sharing that executes Q with the minimized average query latency[2]. The latency of a query $q \in Q$ is measured as the difference between the time point of producing the aggregation result of the query q and the arrival time of the last event that contributed to this result.

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

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

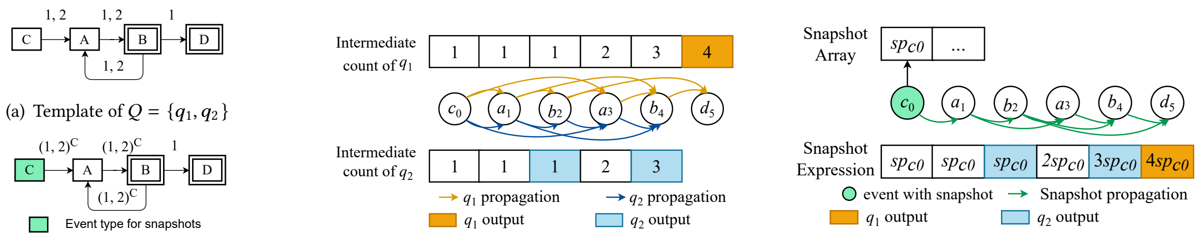
Given a workload Q and its template with k transitions, the number of transition sharing plans for each transition is $O(B_{|Q|})$ in the worst case (Equation 1). Since a workload sharing plan corresponds to the combination of transition sharing plans of all k transitions, the size of the search space S_Π of the workload sharing plan optimization problem Π is $O(B^k_{|Q|})$. Enumerating all possible plans is prohibitively expensive due to its exponential time complexity in the worst case. Hence, an optimizer is needed to efficiently find an optimized workload sharing plan.

3 GLORIA SYSTEM

3.1 Gloria Framework Overview

Figure 4 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 (Sec. 2.2). The optimizer

[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.



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

(b) A sharing plan for Q

(c) Non-shared execution

(d) Shared execution

Figure 3: Non-shared and shared executions over an event stream $I = \{c_0, a_1, b_2, a_3, b_4, d_5\}$ (best viewed in color).

transforms the query template into a Gloria graph to compactly encode all possible sharing opportunities (Sec. 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 (Sec. 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 (Sec. 5.3 and 6). This plan finder features an efficient search time linear in the size of the graph.

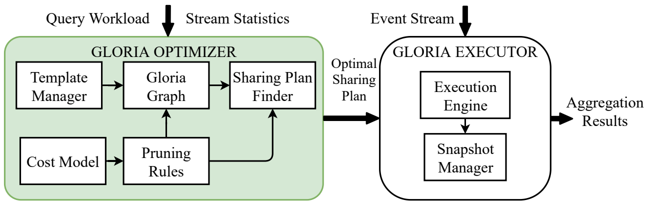


Figure 4: Gloria framework.

The Gloria executor executes the optimized sharing plan produced by the Gloria optimizer. It incrementally computes trend aggregates by propagating intermediate aggregates from previously matched events to new events. It is supported by the snapshot manager that maintains the values of intermediate aggregates per query, called *snapshots*, and shares their propagation. Details of execution are below, while the optimizer is described in Sections 5 and 8.2.

3.2 Gloria Executor

The Gloria executor adopts state-of-art online aggregation methods [26, 27] to support both non-shared and shared executions. Figure 3 features an example template. We use COUNT(∗) as 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 is of the end type of q, this aggregate is output as the final result of q.

Example 3.1. Figure 3(c) shows the non-shared execution of Q in Figure 3(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 the subsequent events, the intermediate aggregates are obtained by summing the values of their predecessors following the propagation for each query. Finally, when d_5 arrives, its aggregate is output as the final count ($fcount = 4$) for q_1.

The execution costs of q_1 and q_2 lie 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 3(c), the non-shared execution costs of all events of type A for $Q = \{ q_1, q_2 \}$ is 6 (i.e., two arrows pointing to a_1 and four arrows pointing to a_3).

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

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

where $|E|$ and $|E_p|$ denote the number of E events and the predecessor events 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 3(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* [26] to support efficient sharing. For a set of shared queries, snapshots are assigned with an event type. The executor creates a snapshot for each event of the event type, which stores the intermediate values of the event for the shared queries, so that the snapshot can be propagated once for all queries. To support propagating snapshots, each shared event maintains a snapshot expression. Intuitively, a shared event summing the snapshot expressions of its predecessors to obtain its own snapshot expression once for all shared queries, instead of summing values for each query. During the shared execution, the snapshot expression of an event e is only evaluated when the event type of e is an end event type for any $q \in Q$.

Example 3.2. Figure 3(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 3(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 is the end event type of q_2, b_2's 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 maintenance are relatively minimal. The former is linear in the number of queries, meanwhile the latter is constant 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 3(d). The evaluations of expressions are needed for the end events b_2, b_4 and d_5. This sharing plan saves 3 propagations compared to the non-shared execution. However, it requires additional evaluations of the snapshot expressions needed for sharing. Hence the sharing plan can only be beneficial when savings from the snapshot propagation outweigh the expression evaluation overhead. Intuitively, a sharing benefit is substantial the more queries and events are shared.

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

$$\begin{aligned} Cost_{share}(E,Q) &= \sum_{E_p \in pe(E,q)} |E_p| \times expr(E_p,Q).len \\ &= \sum_{E_p \in pe(E,q)} (E_p == E_s)?|E_p| : |E_p| \times |E_s| \end{aligned} \tag{3}$$

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 E_s of the snapshots. Notice that such E_s could be E_p or even earlier event type as C for B, D in the above example. Detailed discussion is in Section 5.1.

In general, given an event type E, the snapshot expressions of events of E can be evaluated with evaluation cost:

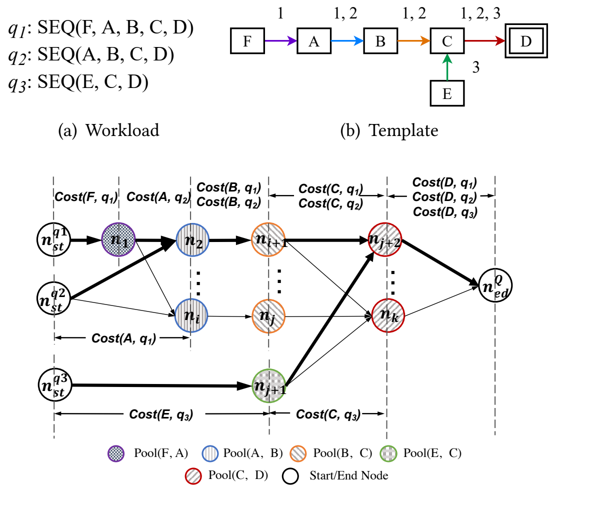
$$Cost_{eval}(E,Q) = \sum_{q \in Q} |E| \times |E_s| \tag{4}$$

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, 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 of E_j for $\bar{Q}$ ($Cost(E_j, \bar{Q})$), when applying the transition sharing plans n_k and n_m.

Example 4.2. Figure 5 depicts the template of a workload $Q = \{q_1, q_2, q_3\}$ and its corresponding Gloria graph, where $q_1 = \text{SEQ}(F, A, B, C, D)$, $q_2 = \text{SEQ}(A, B, C, D)$ and $q_3 = \text{SEQ}(E, C, D)$. 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 $n_m \in Pool(B, C)$.



(c) Gloria graph

Figure 5: Gloria graph model for Q (best viewed in color).

Intuitively, a Gloria path 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 P (or a path P in short) 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.

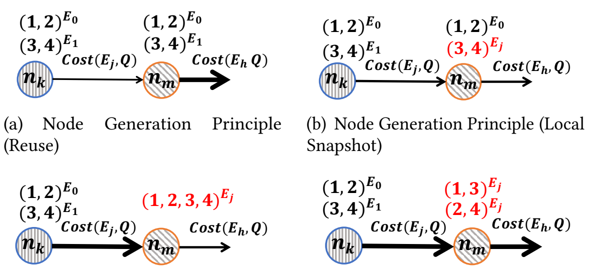
Lemma 4.4. *The weight $P.w$ of a Gloria path P corresponds to the execution cost of the workload sharing plan that P represents.*

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 one workload sharing plan which contains a set of nodes, cannot be represented as a path in the Gloria graph. This means in the node set, 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 minimal path weight in the Gloria graph. $\bar{P}$ corresponds to the optimal workload sharing plan.*



(a) Node Generation Principle (Reuse)

(b) Node Generation Principle (Local Snapshot)

(c) Node Generation Principle (Merge)

(d) Other Nodes

Figure 6: Node generation (best viewed in color).

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 *Kleene* sub-patterns in Section 6.

5.1 Node Generation

When sharing plans of consecutive transitions share different queries, the executor constantly applies evaluation to adapt to sharing plans with different *Q-set*s. Therefore, instead of enumerating nodes independently for each pool, Gloria optimizer generates a node $n_m \in Pool(E_j, E_h)$ from a node $n_k \in Pool(E_i, E_j)$. Otherwise, the executor keeps evaluating for different transition sharing plans, on 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 these plans, which jeopardizes the sharing benefit.

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. For all nodes in $Pool(E_j, E_h)$ that can be generated from n_k, if the weights of the incoming and outgoing edges of a node n_m are known to be larger than other nodes in the same pool. Such local expensive n_m without potential saving opportunities does not 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 edge 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 both case 1 and case 2, when generating such $n_m \in Pool(E_j, E_h)$, 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. Due to the limited space, the full version of proof can be found in our technical report [10]. □

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 $edge(n_k, n_m).w$ in the pool, when it reuses all *Q-set*s and snapshots from n_k. Such node avoids the expensive evaluation for E_j.

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 of 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 $edge(n_k, n_m)$ is the shared execution cost of E_j for $\bar{Q}_1, \bar{Q}_2$.

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

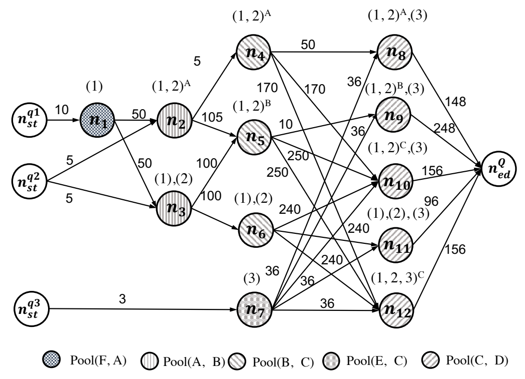
Alternatively, instead of minimizing the weight of the incoming edge, a node $n_m \in Pool(E_j, E_h)$ with heavier incoming edges may have a lighter outgoing edge, which corresponds to lower execution cost of E_h. According to Equation 3, the saving from sharing comes from two aspects. Either sharing the same *Q-sets* with fewer snapshots which shorten the expression length, or sharing with fewer but larger *Q-sets*. 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.

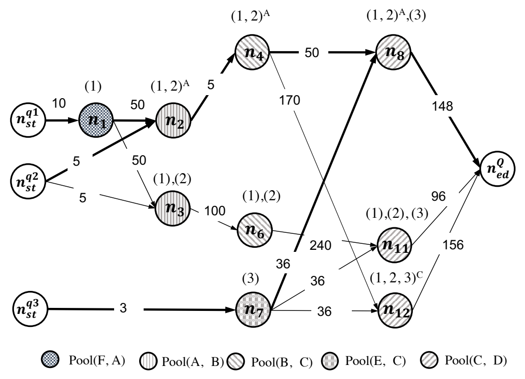
With respect to the first saving opportunity, $n_m \in Pool(E_j, E_h)$ can choose to share the same *Q-set* with n_k but create local snapshots of E_j, if E_j has a lower frequency than the event type of existing snapshots.

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 $\bar{Q}_2 = (3, 4)$. During execution


(a) Full Gloria Graph


(b) Pruned Gloria Graph With Optimal Path

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

Table 3: d_s of nodes.

Figure 7: Gloria Graph with and without pruning.

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 and the value is stored into a new local snapshot of E_j. The weight of $edge(n_k, n_w)$ is the shared execution cost of E_j for $\bar{Q}_1, \bar{Q}_2$, plus the evaluation cost of E_j for $\bar{Q}_2$.

Compared with n_m in Figure 6(a), n_m in Figure 6(b) has a heavier incoming edge 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-set*s. According to our cost model, the weight of $edge(n_k, n_m)$ is the sharing cost plus the evaluation cost of E_j for both $\bar{Q}_1$ and $\bar{Q}_2$.

Compared with the other two principles in Figure 6(a) and Figure 6(b), $edge(n_k, n_m)$ in this case has the heaviest weight, but n_m may have the minimum weight of outgoing edge.

We generate nodes for $Pool(E_j, E_h)$ based on above three principles. Given a workoad Q, If there are multiple $Pool(E_i, E_j)$ that $E_i \in pe(E_j, q), q \in Q$, 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 [10].

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 n_{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 search. Figure 7(b) shows the actual Gloria graph we generate without losing optimality.

Given a node n_m, when there are multiple paths from start nodes to n_m, only the optimal path with the minimum weight needs to be kept, other paths can be safely pruned. The path is stored in the node for further searching, as well as the weight, which presents the distance from all start nodes to n_m, denoted as $n_m.d_s$. Let $n_m \in Pool(E_j, E_h)$ be a node, the distance of n_m to all start nodes can be computed by:

$$n_m.d_s = \sum_{E_i} min\{n_k.d_s + edge(n_k, n_m).w\}, \qquad (5)$$
$$n_k \in Pool(E_i, E_j), E_i \in pe(E_j, q), q \in Q$$

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 5, 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. For n_5, there are two edges from $Pool(A, B)$, $edge(n_2, n_5)$ and $edge(n_3, n_5)$. By Equation 5, $n_2.d_s$+$edge(n_2, n_5).w$ = 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. This lemma can be proved by contradiction. Assume the optimal path $\bar{P}$ passes node $n_m \in Pool(E_j, E_h)$, through an edge $edge(n_0, n_m)$ that doesn't follow Equation 5. By replacing $edge(n_0, n_m)$ with the edge $edge(n_1, n_m)$ following Equation 5, we prove that the new $\bar{P}$ has a lower weight. The full version proof is in our technical report[10]. □

After edge pruning, we consider node pruning for a pool. Given two nodes n_k and n_m in the same pool, if n_m is known to have larger distance to start nodes d_s, as well as a larger distance to 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. First we define for which nodes their distances to end nodes d_e are comparable.

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

Given two comparable nodes, we can estimate the relative magnitude of d_e 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 in Gloria graph.*

Lemma 5.11 can be proved by applying the cost model. Due to the limited space, the proof of Lemma 5.11 can be found in the technical report [10]. 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 $(1, 2)^A$ is lower than $(1, 2)^B$. Therefore, n_5 can be 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 search algorithm on such pruned graph to find the optimal path.

5.3 Path Search 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.

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 [10].

Algorithm 1 Gloria PathSearch

```
Input: Gloria graph $G$
Output: The optimal path $\bar{P}$
1: $\bar{P} \leftarrow$ An empty edge list, $minPathWeight \leftarrow +\infty$
2: if $G.getEndNodes().size = 1$ then // case 1
3:     $endNode \leftarrow G.getEndNodes().get(0)$
4:     edgePrune($endNode$)
5:     $\bar{P} \leftarrow$ GetPaths().get(0)
6: else
7:     $onePath \leftarrow$ true
8:     for each $endNode \in G.getEndNodes()$ do
9:         if $endNode.getIncomingeEdges().size > 1$ then
10:            $onePath \leftarrow$ false
11:    if $onePath$ = true then // case 2
12:        $\bar{P} \leftarrow$ GetPaths().get(0)
13:    else // case 3
14:        reverseEdges($G$)
15:        for each $n \in G$ do
16:            edgePrune($n$)
17:        for each $path \in$ GetPaths() do
18:            if $path.w < minPathWeight$ then
19:                $\bar{P} \leftarrow path$
20:                $minPathWeight \leftarrow \bar{P}.w$
21:            else continue
22: return $\bar{P}$
```

The main algorithm Algo 1 performs in three cases.

Case 1: One End Node (Line 3-5). If there is only one end node in the graph as in Figure 7(b), the optimal path can be selected by simply applying edgePrune 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 12). 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 14-21). Since a node could have multiple outgoing edges, the number of paths could be exponential in the number of edges. 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 its minimum distance to start nodes. With the full graph, we can also maintain one path from all end nodes to n by computing its minimum distance to end nodes $n.d_e$. Therefore, Line 14 reverses all edges in G. Line 15-16 prune the incoming edges for each node to maintain its minimum distance to the end nodes. After that, for each node n, there is only one Gloria path that passes it. Let N be the number of nodes in G, the number of possible paths is reduced to $O(N)$. Line 17-21 enumerate all paths and selects the optimal one with minimum path weight. At last, Line 22 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 is $O(N+T)$. Since each node only maintains one path, the complexity of enumerating all paths 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

Given a *Kleene* sub-pattern SEQ$(A,B)+$ for a workload $Q=\{q_1,q_2\}$ shown in Figure 8(a), a transition from B to A, referred to as ***feedback Kleene transition***, is introduced to the template. To this end, we isolate the cycle from other parts of the template and build a sub-graph for it, called ***Kleene sub-graph***. Note that the *Kleene* sub-graph can be concatenated to an existing Gloria sub-graph of preceding sub-patterns and be extended with subsequent sub-patterns. We pruned *Kleene* sub-graph considering both the concatenation and the extension direction. Our following assumptions assure that there is only one direction of concatenation and extension in a Gloria sub-graph. For multiple directions, we refer to our technical report [10].

Assumptions. To focus on core concepts, we assume that (1) a *Kleene* sub-pattern can only be shared by a workload Q if all $q\in Q$ contains it. and (2) one event type only appears once in a query. We refer to our technical report [10] for a generalized discussion.

6.1 Flat *Kleene* Patterns

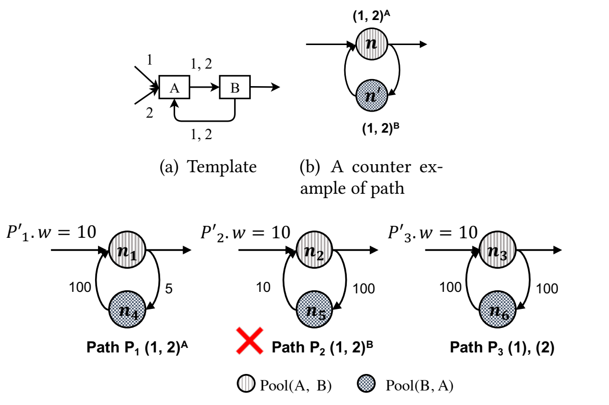
According to our Gloria graph model, two nodes from consecutive pools are connected. In Figure 8(a), there are edges from nodes in $Pool(A,B)$ to nodes in $Pool(B,A)$ and vice versa in the template. These edges create a cycle in the *Kleene* sub-graph, which corresponds to sharing plans on each transition 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.*

We now describe the intuition of Lemma 6.1. The full version of proof can be found in our technical report[10]. *Node Generation Principle (Local Snapshot)* reveals sharing benefits when the event type of new snapshots has lower frequency than the old ones. However, since the cycle structure, the propagation runs in a cycle too, which involves not only snapshot replacing from high-frequency event type to low-frequency event type, but also vice versa. In Figure 8(b), even $|A|<|B|$, replacing the snapshots of A to B introduces evaluation for both transitions and is always more expensive than a consistent sharing with snapshots of A.

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 $Pool(E_i,E_{i+1})(0\le i<k)$ and one pool of



(a) Template (b) A counter example of path

(c) Paths in Gloria sub-graph

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

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.

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 distance of n_k to start nodes $n_k.d_s$ as follows:

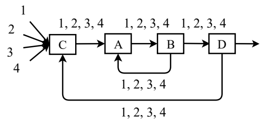
$$n_k.d_s=P'_i.w+P_i.w \tag{6}$$

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

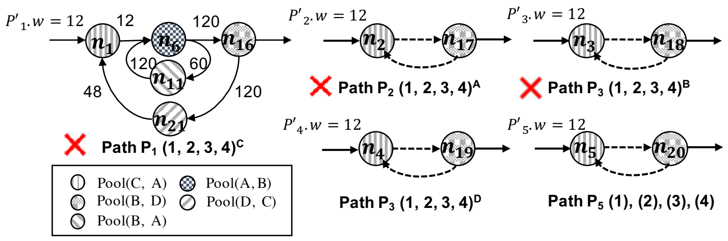
Example 6.3. Continuing Example 6.2, assume $Q=\{q_1,q_2\}$ that both q_1,q_2 start with A, then n_1–n_3 is concatenated to a common start node n^Q_{st}. By applying the cost model in Equation 2, P'_1 to P'_3 have the same weight 10. As the source nodes of extension, $n_1.d_s=115$, $n_2.d_s=120$ and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



(a) Template



(b) Paths in GLORIA sub-graph

Node	Path	d_s
$n_{16}(p)$	P_1	492
$n_{17}(p)$	P_2	252
$n_{18}(p)$	P_3	292
n_{19}	P_4	140
n_{20}	P_5	1204

Table 4: d_s of nodes.

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

LEMMA 6.4. *The path generation principle applies to the Kleene sub-graph of a nested Kleene sub-pattern.*

PROOF. The path of the nested *Kleene* sub-pattern is a nested cycle path. According to Lemma 6.1, every cycle 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). All paths have the same cycle structure so only P_1 is shown in full version, with weight labelled on each edge. Specifically, 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. The edge weights are computed per our cost model in Equation 2, 3 and 4, details can be found in our technical report [10]. We obtain the weights of each path that $P_1.w = 480$, $P_2.w = 240$, $P_3.w = 280$, $P_4.w = 128$ and $P_5.w = 1192$. Assume all queries start with C so that P'_1–P'_5 have the same weight 12. Each node n_{16}–n_{20} obtains its d_s per Equation 6, shown in Table 4. Then we apply *node pruning principle* to the nodes in $Pool(B, D)$ and prune n_{16}–n_{18}.

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

Optimality Discussion. GLORIA optimizer 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, the opportunities of keeping certain *Q-sets* in the concatenated sub-graph are omitted, 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-sets* 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-patterns, changes in cost models and additional aggregation functions.

SEQ vs. *Kleene* Sub-patterns. The sequential node generation (Sec. 5.1) focuses on the long-term benefit but introduces optimization dependencies. In particular, SEQ sub-patterns introduce non-cyclic dependencies and *Kleene* sub-patterns introduce cyclic dependencies. The co-existence of non-cyclic and cyclic dependencies brings additional challenges to the sharing optimization. To solve this problem, we make an assumption in Section 6 that a *Kleene* sub-pattern can only be shared by a workload Q if it is contained by each $q \in Q$, which allows the isolation between the SEQ and *Kleene* sub-patterns, as well as the isolation of different *Kleene* sub-patterns. Under this assumption, GLORIA optimizer finds an optimized sharing plan for the *Kleene* sub-patterns, which always satisfies the cyclic dependencies. Without the above assumption, sharing can potentially happen to any overlapping part among arbitrary sub-patterns, which introduces tangled dependencies. Thus, it is left for future work.

Changes in Cost Model. Handling changes in our cost model can be broken into two parts. The first part, including the edge pruning rule (Sec. 5.2) and the path search algorithm (Sec. 5.3), concerns the optimization on the weighted GLORIA graph (once established). 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 on the edges, it would not affect the correctness of the pruning and path search algorithms. The second part, including the node generation rules (Sec. 5.1) and the node pruning rule (Sec. 5.2), concerns the construction of the GLORIA graph. This part holds independently of the cost model under the assumption that the shared execution costs are proportional to: (1) the length of the snapshot expressions, and (2) the number of *Q-sets* .

Complex Aggregation Functions. Associativity and commutativity of aggregation functions are at the core of the online aggregation methods in both traditional databases [17] and streaming systems [21, 26, 28, 29]. To support non-associative/non-commutative aggregation functions, existing approaches [13, 34] usually choose to store all tuples to be aggregated, which may cause the operator state to grow prohibitively large. To avoid this state explosion, Flink [3] provides a parameter of an appropriate state time-to-live (TTL). However, this might affect the correctness of the query result [4]. Custom optimization for aggregation execution of non-associative/non-commutative functions is an orthogonal and largely open problem in the literature. 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 reports the average of 15 runs.

Data Sets. We evaluate Gloria using three real-world data sets.

• *NASDAQ Stock data set (Stock)* [5] contains stock price history of 20 years. Each record represents an event with a company identifier, timestamp (minutes), open and close price, highest and lowest price and trading volume. Event types correspond to the 3258 unique company identifiers.

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

• *Dublin Bus GPS data set (Bus)* [1] consists of GPS records of buses in Dublin collected by Dublin City Council in 2013. Each record is a timestamped tuple (microseconds) with line id, vehicle journey id, congestion indicator, coordinates and delay time. The vehicle journey id is 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 workloads on each data set.

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

• *Kleene workload* has queries with one shareable flat or nested *Kleene* and 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 such realistic workloads, where the above SEQ and *Kleene* queries appear in one workload. The ratio of *Kleene* and SEQ queries ranges from 1:6 to 1:2.

Methodology. We evaluate the Gloria Optimizer and the execution of plans it generates. We compare our Gloria Optimizer to the following *two alternate optimization approaches*:

• *Greedy Optimizer (Greedy).* For each pool, the *Greedy* optimizer considers the transition sharing plan with minimum incoming edge weight (Sec. 5.1), thus may miss sharing benefits. With respect to node generation, it applies *node generation principle (Reuse)* for sharing, or chooses not to share, based on incoming edge weight. As it only generates one node for each pool, *Greedy* returns the workload sharing plan directly without pruning or path search.

• *Gloria Optimizer without Pruning (NoPrune).* To evaluate the *effectiveness* of pruning rules, we compare the Gloria Optimizer with and without our pruning rules (Sec. 5.2). The later generates a full Gloria graph as in Figure 7(a) that includes all cycle paths in 9(b). This is expensive to construct. After 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 methods:

• Sharing plans generated by a *Greedy Optimizer.*

• Greta[27], a state-of-the-art non-shared method supporting online aggregation over nested *Kleene* patterns.

• Hamlet[26], a state-of-the-art shared online aggregation method equipped with a dynamic optimizer for flat *Kleene* patterns on bursty streams.

Metrics. For query optimization, we measure the *Optimization Time* in milliseconds as the average time difference between the time of receiving the input workload versus producing the sharing plan. This includes the duration of template construction, graph construction, and path search. *Peak Memory* is the maximal memory consumed during graph construction and path search.

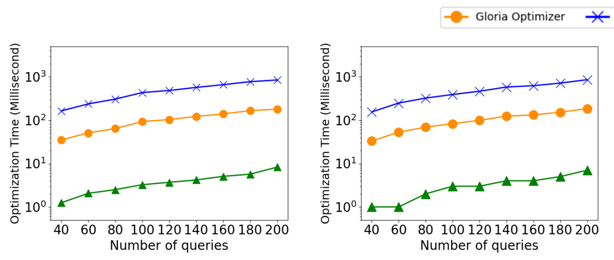
For query execution, we use *Latency* in seconds as average time difference between time of producing aggregation results for a query and arrival time of last relevant event. *Throughput* is the average number of events processed by all queries per second.

8.2 Gloria Optimization Evaluation

SEQ Workload. To evaluate the effectiveness of pruning rules on SEQ patterns, we measure the optimization time of the three optimizers on *Bus* and *Taxi* data sets in Figure 10 while varying the number of queries in the SEQ workload from 40 to 200. On both data sets, Gloria optimizer consistently outperforms *NoPrune* by a factor of 5 to 7, and is slower than *Greedy* by 1.2 order of magnitude. While the *Greedy* optimizer is the fastest among the three, it cannot guarantee the quality of the selected plan. In contrast, both *NoPrune* and Gloria optimizer return the optimal sharing plan for our workloads. Compared to *NoPrune*, Gloria pruning rules reduce the optimization time significantly. Such time savings come from both graph construction and path search. During graph construction, the pruning rules reduce the number of nodes in a pool, thus reducing number of generated nodes in succeeding pools (Sec. 5.2). Given the resulting graph is much smaller, path search runs faster to find the optimal path. In summary, Gloria optimizer efficiently produces the same optimized sharing plan as *NoPrune.*

***Kleene* Workload.** To evaluate the effectiveness of pruning rules on *Kleene* patterns, we compare three optimizers on the *Kleene* workload with 100 queries while varying the length of *Kleene* sub-patterns from 2 to 10 (Figure 11). Gloria consistently outperforms *Greedy* and *NoPrune* optimizers. Since *Greedy* optimizer only maintains one node in each pool, it is the fastest. As the length of *Kleene* sub-patterns increases, the performance difference between Gloria and *NoPrune* increases from 2-fold to 13-fold.

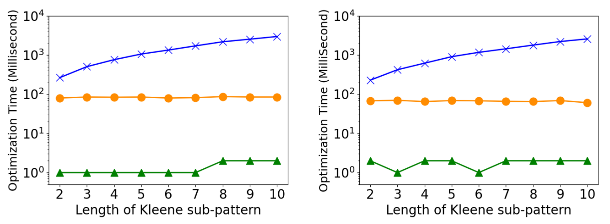
Such performance difference is primarily due to the path consistency property in Lemma 6.1. It ensures that the increasing length of *Kleene* pattern does not increase the number of cycle sharing plans. Hence for a given *Kleene* workload, the size of the generated sub-graph for *Kleene* sub-patterns 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 following SEQ sub-patterns, the size of Gloria graph could still grow exponentially in the worst case. This is consistent with the SEQ



(a) Bus data set

(b) Taxi data set

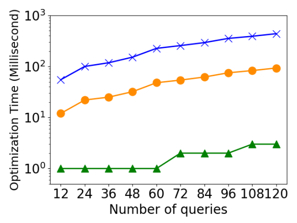
Figure 10: Varying # queries (SEQ).



(a) Bus data set

(b) Taxi data set

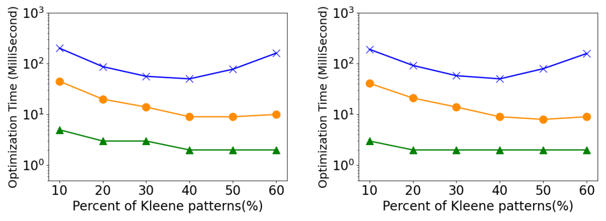
Figure 11: Varying length of *Kleene* sub-patterns (*Kleene*).



(a) Bus data set

(b) Taxi data set

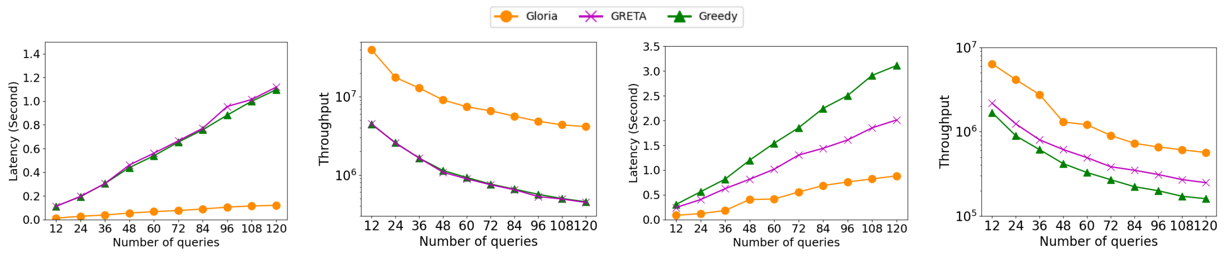
Figure 12: Varying # queries (Mixed).



(a) Bus data set

(b) Taxi data set

Figure 13: Varying percent of *Kleene* queries (Mixed).



(a) Latency (Bus)

(b) Throughput (Bus)

(c) Latency (Stock)

(d) Throughput (Stock)

Figure 14: Execution with different sharing plans (Mixed).

workload. Pruning expensive nodes early on prevents the graph from exploding, benefiting graph construction and path search.

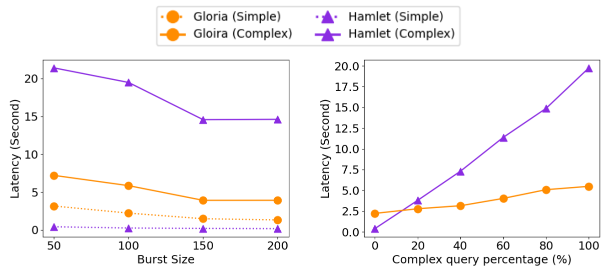
Mixed Workload. Figure 12 compares the three optimizers on mixed workloads with a SEQ-to-*Kleene* ratio as 6:1 with a varying number of queries. Again, Gloria optimizer outperforms *NoPrune* optimizer by 5-fold, while being slower than *Greedy* optimizer by 1 order of magnitude. In Figure 13, we compare 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* drops since the *Kleene* queries reduces the percent of SEQqueries which lower optimization complexity for SEQ queries. Since the portion of *Kleene* queries is still low, the optimization complexity for *Kleene* queries is also limited. When the percentage of *Kleene* queries is larger than 40%, the optimization time is dominated by optimizing the shared *Kleene* queries. In contrast, thanks to the pruning rule applied to the *Kleene* sub-graph (Sec. 6), the optimization time of Gloria optimizer decreases as the percent of *Kleene* queries increases. Our Gloria optimizer only consumes 25% memory of the *NoPrune* optimizer on both data sets. Due to the limited space, we place the results on memory consumption into our technical report[10].

8.3 Gloria Runtime Evaluation

To assess the quality of sharing plans by Gloria, we evaluate several mixed workloads over *Bus* and *Stock* data sets in Figure 14. Greta executor runs each query independently without any sharing. The *Greedy* and Gloria sharing plans are those returned by *Greedy* and Gloria optimizers, 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 14(b) and 14(d). Gloria sharing plan outperforms *Greedy* and Greta plans by 10-fold and 3-fold, respectively. This performance gain is due to Gloria's selection of sharing opportunities. Specifically, Greta plans are not shared, and *Greedy* optimizer fails to harvest all beneficial sharing opportunities since it aggressively reuses existing snapshots instead of consider introducing new ones. Gloria optimizer evaluates costs and benefits of different selections of snapshots and picks the beneficial ones. This way, Gloria optimizer discards expensive while keeping beneficial plans. When the query number increases from 12 to 120, the execution latency of Gloria plan achieves 77-91% and 68-93% speed-up compared to *Greedy* and Greta plans, respectively.

We observe that Greta outperforms (Figures 14(c)) or performs similarly (Figure 14(a)) to *Greedy* sharing plan. This emphasizes the drawback of the greedy strategy and the importance of optimization. If no sharing is the local optima for a transition, then the *Greedy*



(a) Simple and Complex workload (b) Varying complex query percentage

Figure 15: Gloria versus Hamlet.

plan performs similar to GRETA. If the *Greedy* optimizer selects an event type with high frequency for snapshots, these snapshots will be reused for many following transitions. The overhead of summing long snapshot expressions could outweigh the benefit of sharing – akin to 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 non-beneficial sharing.

GLORIA vs. HAMLET. For a fair comparison between HAMLET and GLORIA, we follow HAMLET's bursty stream assumption and use bursty streams. To examine the optimization of HAMLET and GLORIA, we generate a simple and a complex workload [11], 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 (e.g., SEQ(A, B+, C)). In the complex workload, each query is also a flat query, with a *Kleene* operator on a single event type, of length 8 with different predicates.

In Figure 15(a), we measure the latency of the two methods for both simple and complex workloads, while varying the burst sizes of the stream. With an increasing burst size, the latency of both methods decreases on both workloads, because reaching to predecessors becomes less expensive in a bigger burst of the same event type. The experiment result shows that HAMLET wins on the simple workload, but GLORIA consistently outperforms HAMLET by a factor of 3 over the complex workload. With respect to the sharing plan optimization, GLORIA analyzes all the sharing opportunities, meanwhile HAMLET only optimizes for the *Kleene* sub-pattern like *B*+. Therefore, GLORIA is able to harvest more optimization opportunities embedded in the longer patterns. 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, which is guaranteed to be cheaper than a non-sharing plan.

We further measure the latency of these two methods on the bursty stream with a burst size of 100, varying the percentage of complex queries in the workload in Figure 15(b). Similarly, HAMLET outperforms GLORIA when there is no complex queries. With the percentage increases from 20% to 100%, thanks to the global GLORIA static optimizer, GLORIA outperforms HAMLET by 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 [12, 14, 35, 36]. Others employ tree-based models [24]. Some approaches study lazy match detection [22], compact event graph encoding [25], and join plan generation [20]. We refer to the recent survey [15] 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 [27, 28, 30, 36]. A-Seq [30] introduces online aggregation of event sequences, i.e., sequence aggregation without sequence construction. GRETA [27] extends A-Seq by Kleene closure. Cogra [28] further generalizes online trend aggregation by various event matching semantics. However, none of these approaches address the challenges of multi-query workloads, which is our focus.

CEP Multi-query Optimization. Following the principles in relational database systems [33], pattern sharing for CEP have attracted considerable attention. RUMOR [18] defines rules for merging queries in NFA-based RDBMS and stream systems. E-Cube [23] inserts sequence queries into a hierarchy based on concept and pattern refinement relations. SPASS [31] estimates the benefit of sharing for event sequence construction using intra-query and inter-query event correlations. MOTTO [37] applies merge, decomposition, and operator transformations to re-write pattern matching queries. Kolchinsky et al. [21] combine sharing and pattern reordering optimizations for NFA-based and tree-based query plans. Recently, HAMLET [26] adaptively makes sharing decisions at run time based on the current stream properties for flat *Kleene* queries.

However, these approaches [21, 31, 37] do not all support online aggregation of event sequences, i.e., they instead construct all event sequences prior to their aggregation. This degrades query performance. To the best of our knowledge, SHARON [29], Muse [32], and HAMLET [26] are the only solutions that support shared online aggregation. However, SHARON does not support Kleene closure, MCEP and COGRA [21, 28] 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 result in sub-optimal sharing plans for event trend aggregation queries, as sharing opportunities can be 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.

ACKNOWLEDGMENTS

This work is supported by NSF grants IIS-1815866 and IIS-1852498.

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://github.com/LeiMa0324/Gloria, 2022.

[10] Gloria: Graph-based sharing optimizer of multi-query event trend aggregation. https://github.com/LeiMa0324/Gloria/blob/master/Gloria_Technical_Report.pdf, 2022. Technical report.

[11] Gloria vs. hamlet experiment workloads. https://github.com/LeiMa0324/Gloria/tree/master/src/main/resources/stock/GloriaVSHamletWorkload/MixWorkload, 2022.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[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] 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.

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

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

[26] 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.

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

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

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

[30] 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.

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

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

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

[34] 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.

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

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

[37] 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.