Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Adaptivity in Distributed Event Processing with Multi-Node Query Placement

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Kristina Pianykh
geboren am: 28.07.1995
geboren in: Twer

Gutachter/innen: Prof. Dr. Matthias Weidlich
Prof. Dr. rer. nat. Lars Grunske

eingereicht am: ..... verteidigt am: .....

Abstract

Distributed event processing systems are widely used in various applications, including sensor networks, IoT, and cloud computing. One of the challenges imposed by such systems is managing data transmission costs while ensuring efficient event processing. This thesis introduces an adaptive strategy for networks with multi-node query placement to address scenarios where fluctuations in event rates lead to high transmission costs and risk network overload.

Contents

1 Introduction

Complex Event Processing (CEP) is a computational paradigm designed to identify patterns or meaningful events within streams of real-time data. Unlike traditional database systems that rely on static data, a CEP system continuously processes data in motion, reacting to events as they occur. This technology shows great potential across various domains, such as financial markets (Teymourian et al. [2012]), Internet of Things (IoT) (Lima et al. [2022], Shah et al. [2022]), fraud and security attack detection (Roldán et al. [2020], smart grid management (Zu et al. [2017]), and crisis-management systems (Paraiso et al. [2012]), where real-time decision-making is critical.

CEP works by defining and applying patterns, or queries, to a stream of events. These patterns can represent conditions or sequences of interest, such as a specific combination of stock prices or sensor data. Over the years, CEP has evolved into an essential technology, especially in distributed systems where events are generated across multiple sources, such as in cloud environments or sensor networks (Cugola and Margara [2012]). Its relevance is further emphasized by the growing need to process large volumes of data in real-time and derive actionable insights from them.

One important distinction of various CEP methods is between centralized and distributed query evaluation. In the centralized approach, events are gathered and processed at a single location, which limits scalability due to network overhead and single-point bottlenecks. On the other hand, a distributed model pushes parts of the processing (i.e. CEP operators) toward the sources of input generation so that events are evaluated across multiple nodes, which improves scalability and resilience by reducing transmission costs and making use of local compute resources (Akili et al. [2023b], Flouris et al. [2020]).

One such approach is represented by the INEv framework, which proposes in-network evaluation (INEv) graphs that decompose and distribute query evaluation across multiple nodes, allowing more fine-grained control over the processing of events (Akili et al. [2023b]). Unlike traditional methods, where sub-queries are processed at a single node, INEv enables sub-queries with multi-node placements, exploiting partial results from one sub-query in others. This method significantly reduces transmission overhead and makes pattern matching truly distributed, addressing the limitations of existing frameworks that still rely on centralized components or do not make use of query-rewriting and efficient multi-node placement such as Flink, Esper, or Apache Strom (Apache Software Foundation [2019a], Inc. [2024], and Apache Software Foundation [2019b], respectively).

One of the key challenges highlighted in the INEv graph approach is the system's ability to adapt to fluctuations in event rates or distribution patterns. INEv graphs are initially constructed with a static evaluation plan based on the assumption that event rates and distribution remain stable over time. However, in dynamic, distributed environments, such stability is often difficult to achieve due to factors such as varying network performance and changes in event frequencies. Violation of certain assumptions underlying the construction of an INEv graph can cause an increase in transmission overhead. To address this issue, the authors propose two adaptive strategies that allow the system to modify its evaluation plan in response to these changes. One such strategy, which focuses on repairing invalid sub-queries, has been explored in detail in the study by

Palm [2023]. The second strategy, which involves dynamically adjusting query placement from a multi-node to a single-node configuration when the multi-node placement becomes harmful, is the topic of the current study.

In this thesis, I will focus on implementing an adaptive strategy to address the scenario where the multi-node query placement in distributed CEP increases network costs.

The goal of this thesis is to implement an adaptive strategy to address the scenario where the multi-node query placement in distributed CEP increases network costs. This strategy is required to dynamically adjust query placement without the need to pause data processing or restart the engine, ensuring continuous and efficient real-time event processing. A series of experiments will be conducted to examine the optimization potential of this approach. Section 2 provides an overview of In-Network Evaluation (INEv) for CEP. Section 3 focuses on query placement within INEv graphs and also formulates the problem addressed in this thesis, specifically the repair of invalid multi-sink query placements to reduce network costs. Section 4 conceptualizes the adaptive strategy, and Section 5 covers its implementation and the associated design challenges. In Section 6, we outline the experimental design and analyze the optimization potential of the repair strategy based on the experimental results. Finally, Section 7 discusses the limitations of the developed strategy, proposes improvements, and suggests directions for future research. The study concludes in Section 8.

2 Background

2.1 System Model

Before covering the specifics of constructing an INEv graph, it is essential to first establish the context defined by the system model and the query language used in this study. The complete list of the notations used throughout this study can be found in Table 1.

Let $\mathcal{E} = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$ be a set of n events generated in a given system. An event is defined as a meaningful change or occurrence within a system that provides information about the state or activity of various components and represents discrete units of information. We distinguish between primitive events that represent atomic, non-decomposable units of information and complex events that are composed of other events. We call $\Gamma = (G_{top}, f, r)$ an event network where $G_{top} = (V_{top}, E_{top})$ is a topology graph, $f : \mathcal{E} \to 2^{V_{top}}$ is a function that assigns events to nodes, $r : \mathcal{E} \to \mathbb{R}$ is a function that maps events to rates with which they are generated.

G_{top} describes a connected directed graph with nodes V_{top} and edges E_{top}. Events are sent along the edges E_{top} and a hop describes the transition of an event from node v_i to node v_j where v_i, $v_j \in V_{top}$ and $v_i \neq v_j$ along the connecting edge (v_i, v_j). An event rate $r(\epsilon)$ denotes the number of occurrences of events of type ϵ within a time unit. The notation $r(\epsilon_i) \gg r(\epsilon_j)$ for any pair of event types ϵ_i, $\epsilon_j \in \mathcal{E}$ with $\epsilon_i \neq \epsilon_i$ is used to emphasize that the rate of event type ϵ_i is significantly greater than that of ϵ_j. The global rate of an event ϵ is defined as $R(\epsilon) = r(\epsilon) \cdot ||f(\epsilon)||$ which is the sum of all the source nodes supplying the event ϵ multiplied by the local rate of the event. To put it differently,

the global rate represents a cumulative sum of all the local rates of a particular event type.

Table 1: Table of Notations and their Explanations

Notation	**Explanation**
$\mathcal{E} = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$	Event universe of size n
$\Gamma = (G_{top}, f, r)$	Network system where $G_{top} = (V_{top}, E_{top})$ is a topology graph, $f : \mathcal{E} \to 2^{V_{top}}$ is a function that assigns events to nodes, $r : \mathcal{E} \to \mathbb{R}$ is a function that maps events to rates.
$r(\epsilon)$	Local rate of $\epsilon \in \mathcal{E}$
$R(\epsilon) = r(\epsilon) \cdot \|\|f(\epsilon)\|\|$	Global rate of $\epsilon \in \mathcal{E}$
$q = (O, \beta)$	Query q where $O = O_p \cup O_c$ the set of primitive and complex operators and $\beta : O_c \to O^k$ with $k \in \mathbb{N}$ the function that assigns operators $o \in O$ to complex operators
$\sigma(o)$	Selectivity of an operator $o \in O$
$Q = \{q_1, ..., q_n\}$	Query workload which is a set of $n \in \mathbb{N}, n > 0$ queries evaluated within a network
$\lambda(q), \lambda(p)$	Inputs of q or p
Π_q	All possible projections for q
$c(q) = (\mathfrak{P}, \lambda)$	Combination for q where $\mathfrak{P} \subseteq \Pi_q$ is a set of projections that result in a complete and correct evaluation of q and $\lambda : \Pi_q \to \Pi_q^k$ with $k \in \mathbb{N}, k > 1$ a function that assigns projections to k predecessors
$\mathfrak{P}_{np} \subset \mathfrak{P}$	Non-partitioning inputs for a projection p s.t. $\lambda(p) = \mathfrak{P}_{np}$
$p_{part} \in \lambda(p)$	Partitioning input for a projection
$I = (V, E)$	INEv graph with V nodes in the form $v = (v_{proj}, v_{node}, v_{graph}, v_{os}) \in V$ and E edges
$\psi(I)$	Total transmission costs for I
$V_{multi} \in V_{top}$	Multi-sink nodes evaluating a projection with a multi-node placement type
$v_{fallback} \in V_{multi}$	Fallback node selected from V_{multi}

2.2 Query Language

A query q evaluated in a network system is defined as $q = (O, \beta)$ and associated with a time window that imposes evaluation bounds on the otherwise unbounded stream of events. A query can be conceptualized as an ordered tree of operators O as vertices of the tree. We distinguish between primitive operators O_p that map to an event type and composite (complex) operators O_c that are the parent to some ordering of child operators. The function $\beta : O_c \to O^k$ with $k \in \mathbb{N}$ assigns k number of child operators to a composite operator. $O = \{SEQ, AND\}$ is the set of operators under examination in the current study with the following semantics:

- **SEQ** produces a match from the ordered sequence of the child operators $o \in \beta(O_c)$.

For instance, for $k = 3$ and $\beta(O_c) = \{o_1, o_2, o_3\}$, $SEQ(o_1, o_2, o_3)$ constructs a match for the events in the ordered set $\{o_1, o_2, o_3\}$ based on the time of their creation.

- **AND** produces a match from a combination of the child operators $o \in \beta(O_c)$. For example, for $k = 3$ and $\beta(O_c) = \{o_1, o_2, o_3\}$, $AND(o_1, o_2, o_3)$ constructs a match for any ordered variation of the set $\beta(O_c)$.

Note that a primitive operator $o \in O_p$ that references an event type $\epsilon \in \mathcal{E}$ produces a match for each occurrence of ϵ.

The concept of selectivity is introduced to constrain the otherwise greedy event selection strategy where each occurrence of a relevant event type results in a partial match. Selectivity, denoted as $\sigma(o)$ for an operator $o \in O$ and ranging in values from 0 to 1, represents the probability of an event, filtered by its predicate, to be considered a partial match. The selectivity of an operator has a direct impact on the match rate. For a primitive operator $o \in O_p$ with the selectivity $\sigma(o)$, the resulting rate is given by $r(o) = \sigma(o) \cdot r(\epsilon)$. For each of the complex operators in O_c, the selectivity is defined as follows:

- $r(o_c) = \sigma(o) \cdot \Pi_{1 \leq i \leq k} r(o_i)$ for o_c with the SEQ semantics.
- $r(o_c) = \sigma(o) \cdot k \cdot \Pi_{1 \leq i \leq k} r(o_i)$ for o_c with the AND semantics.

The notion of selectivity is essential for simulating a predicate evaluation on incoming events, determining whether each event qualifies as a partial match.

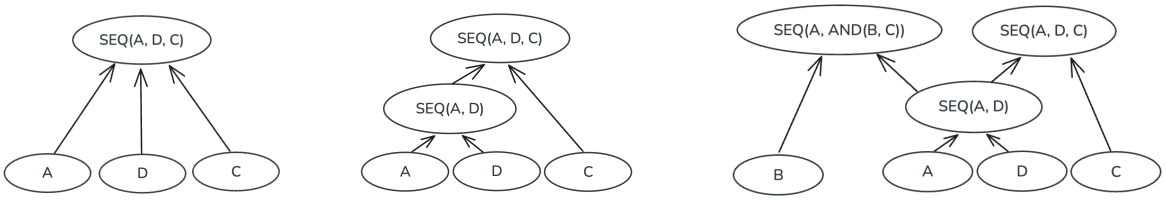
2.3 In-Network Evaluation

Given a network system $\Gamma = (G_{top}, f, t)$ and a query workload Q, the In-Network Evaluation (INEv) attempts to reduce the rate with which events are sent over the network by addressing the three problems:

1. How should Q be split into sub-queries?
2. Given a certain splitting, or combination, of Q how and where should be the resulting sub-queries placed?
3. How should the resulting matches be forwarded within the network?

The first point refers to the mechanism of splitting a query into subqueries, or projections, $p = (O, \beta)$ constructed from a subset of the event types: $\mathcal{E}(p) \subset \mathcal{E}$. A set of all projections for a query q are referred to as Π_Q. A set of projections realizing the correct and complete evaluation of q is referred to as a combination $c = (\mathfrak{P}, \lambda)$ where $\mathfrak{P} \subseteq \Pi_q$ and $\lambda : \Pi_q \to \Pi_q^k$ with $k \in \mathbb{N}$, $k > 1$ a function that assigns projections to its k predecessors in c, also called inputs to p. Projections can be exploited for multiple queries within a query workload. Figure 1 illustrates how a combination can be distinguished from a query tree.

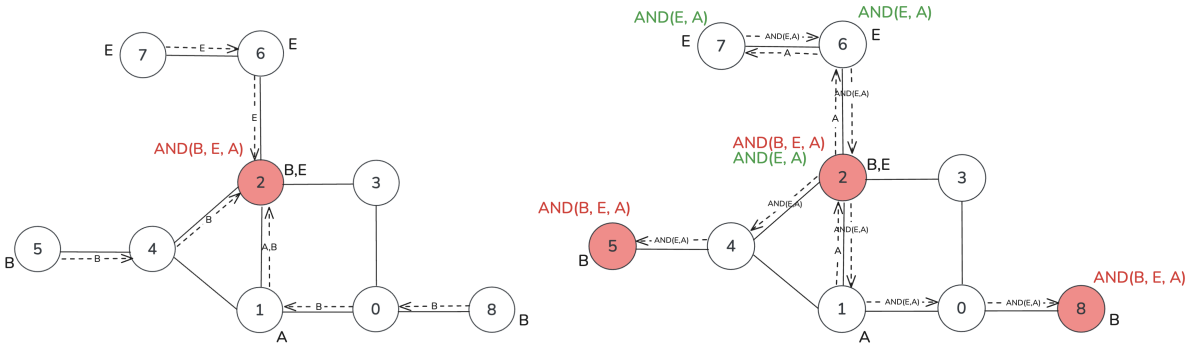
Figure 1: Difference between (a) a query tree for $q = SEQ(A, D, C)$, (b) a combination for q and (c) projection sharing for a query workload $Q = \{q_1, q_2\}$ with $q_1 = SEQ(A, AND(B, C))$, $q_2 = SEQ(A, D, C)$



(a) Query tree (b) Combination (c) Combination for query workload

Let G_{top}^{sub} be a subgraph in G_{top}. An INEv graph is a directed acyclic graph (DAG) $I = (V, E)$ for a given event network $\Gamma = (G_{top}, f, t)$ with the topology graph $G_{top} = (V_{top}, E_{top})$. The vertices V are defined as $V \subseteq \Pi_Q \times V_{top} \times G_{top}^{sub} \times (2^{\mathcal{E}} \cup \{\equiv\})$ and the edges as $E \subseteq V \times V$. $(2^{\mathcal{E}} \cup \{\equiv\})$ is the target domain (or image) of the function $os : \{\Pi_q \rightarrow (2^{\mathcal{E}} \cup \{\equiv\})$ that assigns a projection $p \in \Pi_q$ to a subset of event types to forward on a match for this projection. The symbol $\equiv$ indicates that no events should be forwarded. Figure 2 illustrates an example of such an INEv graph as opposed to centralized query evaluation.

Figure 2: Centralized (left) vs. distributed (right) processing of $q = AND(B, E, A)$

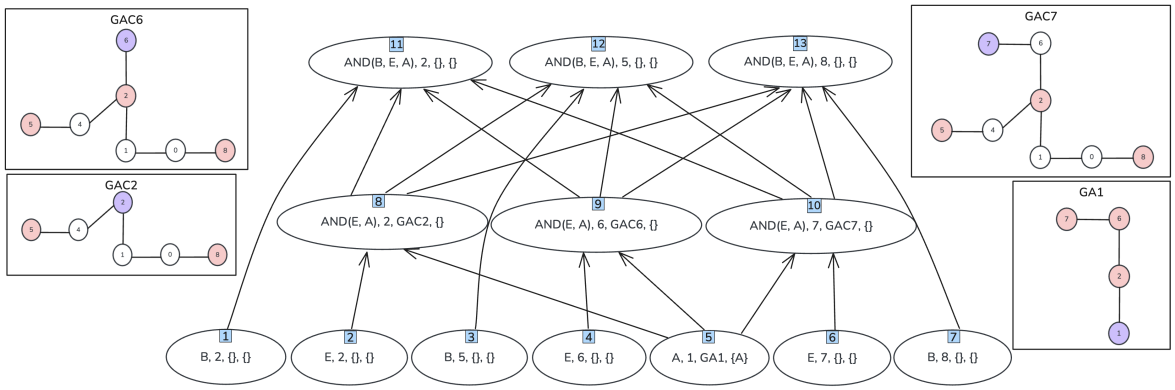


The Figure 2 displays a topology with $V_{top} = \{0, 1, 2, ..., 8\}$ that process the query $q = AND(B, E, A)$ and supply $\epsilon \in \mathcal{E}$ at the nodes $f(A) = \{1\}$, $f(B) = \{2, 5, 8\}$, and $f(E) = \{2, 5, 6, 7\}$. Note that the rates of the input events are distributed in a way such that $r(B) \gg r(E) \gg r(A)$. The centralized model (on the left) represented on the left subgraph, places q on node 2 since it is the most suitable node since it generates two high-frequency inputs for q and has the highest degree of centrality in the network graph. In the distributed evaluation plan generated by the INEv approach (on the right), q has a multi-node placement on nodes 2, 5, and 8, at the source nodes that generate B events so these high-frequency inputs would not be forwarded across the network but be processed locally, instead. Moreover, q is rewritten such that $\lambda(q) = \{AND(E, A), B\}$ where $p = AND(E, A)$ is a projection with $\lambda(p) = \{E, A\}$ placed at the source nodes supplying E events whose rate is higher than that of A. The multi-node placement type

for both q and p was selected intentionally to push the operators, processing them, closer to the sources of high-frequency inputs and by doing so to reduce transmission costs.

The INEv graph $I = (V, E)$ generated for this network system to process the workload Q can be found in Figure 3. Each vertex $v \in V$ has the form $v = (v_{proj}, v_{node}, v_{graph}, v_{os})$ where $v_{proj} \in \mathfrak{P}$ evaluated on node $v_{node} \in V_{top}$, the matches for this projection are forwarded along the associated path in v_{graph} and $v_{os} \subset \mathcal{E} \cup \{\equiv\}$ is the output selector for the projection, i.e. it specifies what events resulting in a match should be forwarded.

Figure 3: INEv graph query workload $Q = \{AND(B, E, A)\}$ in the event network from Figure 2



The projection $p = AND(E, A)$ is evaluated at the nodes supplying E events because $r(E) \gg r(A)$. Events of type A are forwarded to the sinks of p along the paths represented in graph $GA1$. The matches for p are all sent to the sinks of q where B events are generated and processed locally following the paths in graphs $GAC2$, $GAC6$, $GAC7$ since $r(p) \ll r(B)$ with $r(p) = \sigma(p) \cdot 2 \cdot r(E) \cdot r(A)$.

Cost Model

The effectiveness of INEv graph construction for distributed query evaluation is measured in transmission costs (TC) which represents the amount of data (i.e. events) sent across the network per time unit. The benefits of using a distributed evaluation model become apparent when comparing TC in both the centralized and distributed models. In contrast to the centralized model where a query is evaluated on exactly one node, thus, causing all events to be transmitted to the single sink of the query, the INEv approach makes use of the difference in event rates which determines the outcome of query-splitting, as well as the choice of a projection placement type. Specifically, for the event network from Figure 2, the transmission cost for the centralized model amounts to $5 \cdot r(B) + r(A) + 2 \cdot r(E)$, compared to $3 \cdot r(A) + 7 \cdot r(AND(E, A)) + 7 \cdot r(AND(E, A))$ where forwarding of the most high-frequency event B is avoided.

Let $I = (V, E)$ be a constructed INEv graph for a given event network. As defined in 2.3, $v = (v_{proj}, v_{node}, v_{graph}, v_{os}) \in V$ where $v_{proj} \in \mathfrak{P}$ where v_{graph} refers to the forwarding graph $v_{graph} = (V_v, E_v)$ for the matches of v_{proj}. The TC for a vertex v is defined as

$tc(v) = \|E_v\| \cdot r(v_{proj})$, i.e. the match rate for the given projection multiplied by the length of the forwarding paths for each match event. The total TC for the INEv graph I can be, thus, computed as follows:

$$\psi(I) = \sum_{v \in V} tc(v) \tag{1}$$

3 Problem Statement

An INEv graph is constructed under the assumption that the event rates remain the same over time. That is, an evaluation plan for the optimized query decomposition and placement is guided, among other factors, by the rates of the inputs for the queries in question. In the real world, however, the idea of fixed rates in a complex distributed setting is rather far-fetched as network communication is extremely volatile by its very nature. In a real-world application, such systems rarely exist in isolation, instead, they are integrated with data producers which, in their turn, can incur additional latencies and amplify existing fluctuations in message frequencies. How can we adapt to these changes in the context of In-Network Evaluation?

The authors of the INEv framework discuss two potential scenarios when the originally generated evaluation plan can become harmful, i.e. it would inflict higher transmission costs in the system:

1) The output rate of a projection becomes higher than the sum of its input rates.

2) The rate of the non-partitioning inputs of a multi-sink query in a multi-node placement increases.

The first scenario is analyzed and discussed in detail in the unpublished thesis by Palm [2023]. The second scenario formulates the problem for the current study.

A query placement is one of the fundamental issues tackled in INEv graph construction. For a network system $\Gamma = (G_{top}, f, r)$ with $G_{top} = (V_{top}, E_{top})$ and a combination $c = (\mathfrak{P}, \lambda)$, a placement assigns a projection $p \in \mathfrak{P}$ to a set of nodes, i.e. it is a function $\mathfrak{P} \rightarrow 2^{V_{top}}$. One can distinguish between a single-node and multi-node placement type. On the one hand, opting for multi-node placement for a projection reduces the amount of data transmitted over the network and allows nodes to perform the evaluation locally. This is an especially effective strategy when one of a projection's inputs is a high-frequency event. Such input $p_{part} \in \lambda(p)$ is referred to as the *partitioning input*. On the other hand, one must consider the fact that multi-node query placement increases the cost of forwarding the inputs with lower rates to multiple nodes, instead of to what otherwise would be a single sink. The inputs $p' \in \lambda(p)$ with $r(p') \ll r(p_{part})$ are referred to as *non-partitioning inputs* $\mathfrak{P}_{np} \subset \mathfrak{P}$ with $\mathfrak{P}_{np} = \mathfrak{P} \setminus \{p_{part}\}$. It follows that multi-node placement is beneficial when the rate of some event type (to be used as partitioning inputs) is significantly higher than those of others, as high-rate events are no longer sent over the network.

Let $ST(N', G_{top})$ define a Steiner tree for the topology G_{top} in the network $\Gamma = (G_{top}, f, r)$ which spans nodes N'. The Steiner tree is employed as a heuristic for constructing the shortest paths between the potential nodes for a multi-sink placement. In

other words, $\|ST(N', G_{top})\|$ corresponds with the number of edges in this tree. The following inequality formulates the condition for a multi-sink placement:

$$R(p_{part}) > \|ST(f(i), G_{top})\| \cdot \sum_{p' \in \mathfrak{P}_{np}} R(p') + R(p) \tag{2}$$

A multi-node placement becomes beneficial when the global rate of the partitioning input p_{part} is higher than the cost of exchanging the non-partitioning inputs $\mathfrak{P}_{np}$ between the multi-sink nodes V_{multi} given their global rates.

There are two ways in which this inequality can be violated to bring the originally constructed multi-node placement for a given projection into question. Either the rate of the partitioning input drops or the rates of the non-partitioning inputs increase. While the first scenario would not inflate the TC, the second scenario can be deemed potentially harmful since it inflicts the overhead of forwarding the now high-frequency non-partitioning events to the multiple sinks of the projection. We will therefore focus on mitigating this situation and attempt to offer a repair strategy whose purpose is to keep the TC down.

4 Adaptive Strategy

The primary objective of the repair strategy proposed in this study is to mitigate network flooding that results from significant changes in input rates for a given multi-sink query by changing the query placement type from multi-node to single-node. Rather than exchanging inputs between the multiple sink nodes, all inputs are forwarded to the single sink. This approach is expected to reduce the number of hops required for high-frequency, non-partitioning inputs, leading to a reduction in TC. This method, however, introduces a number of challenges. How should the node for a single-node placement of Q be determined? How should forwarding rules be modified? How should these updates be communicated among nodes? Lastly, how can we ensure the correctness of the evaluation during the transition from multi-node to single-node query placement without disrupting or restarting system?

The repair measures required for the seamless transition of the system from multi-node to single node-placement can be broken down into a series of steps described below.

1. Monitor event rates and continuously (re)compute the left-hand and right-hand sides of inequality 2 to track any substantial drifts in event rates. This task is performed by a monitoring component deployed on each node within the network.

2. If the inequality no longer holds, initiate the repair by triggering a central system component referred to as the coordinator.

3. The coordinator computes an updated evaluation plan for the network in the target state. First, it determines the fallback node for single-node query placement. In the next step, it updates the forwarding paths for each node to align with the placement type. Lastly, it computes a timestamp $t1$ in the future that represents

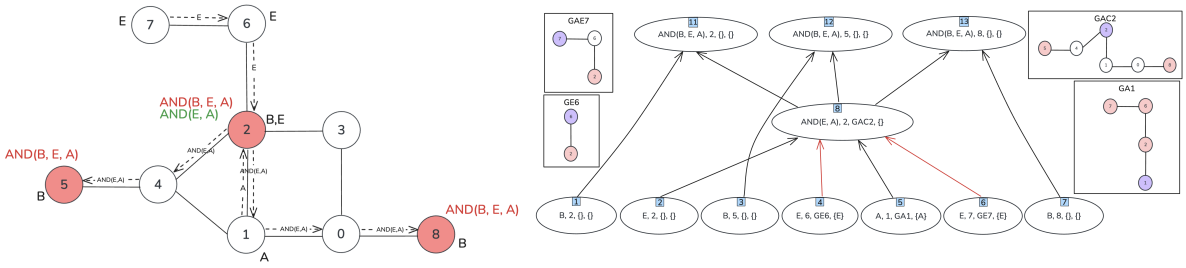
the time for the transition of the system into the target state. The resulting plan, as well as the timestamp, are then broadcast to all the nodes in the network.

4. The nodes save the updated forwarding tables received from the coordinator locally, in preparation for replacing the original forwarding table at the transition time $t1$. The nodes $v \neq v_{fallback} \in V_{multi}$, where $v_{fallback} \in V_{multi}$ is the fallback node, begin buffering the partitioning input for a one-time flush to $v_{fallback}$ at $t1$. These partitioning inputs are then matched on the fallback node with the non-partitioning input events buffered locally, a process we will refer to as retrospective matching. Buffering partitioning input events on nodes $v \neq v_{fallback} \in V_{multi}$ and non-partitioning input events on the fallback node is essential to prevent data loss during the transition and to ensure the correctness of the pattern matching process.

5. On the time $t1$, the nodes apply the instructions received from the coordinator, allowing the system to transition to the state with single-node query placement. The nodes $v \neq v_{fallback} \in V_{multi}$ flush the buffered partitioning events to $v_{fallback}$. The fallback node performs retrospective matching with the locally buffered non-partitioning events on top of the regular query processing.

Figure 4 illustrates the final state of the example network from Section 2.3, derived from the updated INEv graph. The selection criteria for the fallback node are explained in Section 5, which covers the implementation details. In short, the fallback node $v_{fallback} = 2$ was chosen based on its centrality within the graph, as well as the locality of processing of any complex inputs for q if there are any, which in this specific example is the case for $AND(E, A) \in \lambda(q)$. The following modifications were applied to the original INEv graph $I = \{V, E\}$.

Figure 4: Event network (left) and INEv graph (right) with repair



- The forwarding rules for $\lambda(p)$ are adjusted incrementally in a series of steps.
  - First, the rules to send the inputs $\lambda(p)$ to $v = (v_{proj}, v_{node}, v_{graph}, v_{os}) \in V$ where $v_{node} \neq v_{fallback} = 2 \in V_{multi}$ are removed: $E = E \backslash \{(4, 9), (5, 9), (6, 10), (5, 10)\}$.
  - Next, the rules to forward $p \in \lambda(q)$ are disabled: $E = E \setminus \{(9, 11), (9, 12), (9, 13), (10, 11), (10, 12), (10, 13)\}$.
  - Finally, we add the rules to forward the inputs $\lambda(p)$ to $v = (v_{proj}, v_{node}, v_{graph}, v_{os}) \in V$ where $v_{node} = v_{fallbask} = 2$: $E = E \cup \{(4, 8), (6, 8)\}$. The last step is accompanied by recomputing the forwarding paths for each of the newly added edges resulting in graphs $GE6$ and $GE7$, respectively.

- The projection $ANDD(E, A)$ is removed from nodes $6, 7 \in V_{top} : V = V \setminus \{9, 10\}$

For the event network from Subsection 2.3, the INEv graph resulting from applying the repair strategy and its projection onto the network are represented in Figure 4.

The effectiveness of the outlined strategy is evaluated based on the results of experiments carried out on a sample of various event networks in Section 6.

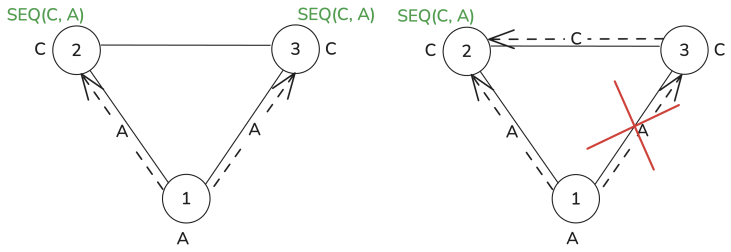
Coordinator

The introduction of the coordinator in 2 adds a centralized element to our system. Although this component could be integrated with one of the nodes in the topology to align with a distributed paradigm, we opted for the centralized alternative for several reasons. To begin with, the coordinator does not engage in data processing or pattern matching. Instead, it manages synchronization between nodes when repair is triggered. Moreover, the coordinator has a comprehensive view of the entire event network. Unlike individual nodes, which are only aware of their immediate neighbors and lack knowledge of the overall network topology, the coordinator has full visibility. The broader visibility scope is crucial because individual nodes, limited to their local perspective, are incapable of formulating an updated evaluation plan for the system's transition to the target state. It is because of its system-wide awareness and understanding of the complete network that the coordinator is uniquely positioned to manage the transition of the system into the target state seamlessly.

Retrospective Matching

It is important to ensure that the outlined strategy will result in correct query evaluation. To understand why buffering partitioning inputs on non-fallback nodes and retrospective matching on the fallback node help to achieve this goal, consider the small network in Figure 5 before and after changing the placement type for $q = SEQ(C, A)$.

Figure 5: Before (left) and after (right) the transition from multi-node placement of $q = SEQ(C, A)$ to a single-node placement type

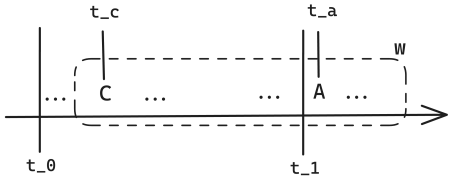


The network system $\Gamma = (G_{top}, f, r)$ features the topology $G_{top} = \{V_{top}, E_{top}\}$ with the nodes $V_{top} = \{1, 2, 3\}$, the edges $E_{top} = \{(1, 2), (2, 3), (1, 3)\}$, the event sources $f(A) = \{1\}, f(C) = \{2, 3\}$ and the rates $r(C) \gg r(A)$. The query $q = SEQ(C, A)$ is, therefore, placed to be processed at the sources of C events due to their high rates at nodes 2 and 3. There is only one source of A's so we forward them from node 1 to the

sinks of q. If A events suddenly increase in their generation frequency to the extent that it overloads the network, it will initiate the adaptive strategy. The query is removed from node 3 (in this particular network it could also be node 2) and the forwarding rules are updated to send the inputs of the query to $v_{fallback} = 2$. Let $t0$ and $t1$ be the timestamps representing the time at which a drift in event rates was detected (i.e. the time when the inequality 2 was violated) and the time for the transition to the single-node query placement, respectively. Let C be generated at node 2 on time t_c s.t. $t_0 \leq t_c \leq t_1$. Let A be generated at time t_a s.t. $t1 < t_a$ (see Figure 6). Then C will not be matched with A if they fall into the same processing window w because C as an instance of the partitioning input was generated and processed locally at node 2 before the transition took place whereas the event A was forwarded only to the fallback node once the system has completed its transition at $t1$.

The scenario described above illustrates the necessity for a mechanism to prevent data loss and ensure the correctness of the query evaluation during and after the transition. One potential solution to this issue would be to buffer the partitioning inputs on $v \neq v_{fallback} \in V_{multi}$ to use for retrospective matching with non-partitioning inputs at $v_{fallback}$. In our specific example, it means that the C should be buffered at node 2 during the phase between $t0$ and $t1$ and then flushed to $v_{fallback} = 2$ on the transition time $t1$ to be combined with the A in order to produce a match.

Figure 6: Example of incorrect matching of C sent at t_c and A sent at t_a for $q = SEQ(C, A)$ within the time window w as a result of data loss during the transition at $t1$



5 System Implementation

The implementation and simulation of the repair strategy were executed on a compute unit with the Apple M2 Pro chip, 10-core CPU, and 16GB RAM. The project with the source code is publicly available [1]. Data streaming and query processing were implemented using Apache Flink [2021] framework in Java (version 1.14.2). The wrapper library developed by Akili et al. [2023a] and built around the existing Flink framework for the study Akili et al. [2023b] on distributed in-network query evaluation, served as the foundation for our implementation. While the library offered a solid starting point, modifications were required to extend its functionality and integrate it with the adaptive strategy. These modifications included fixing bugs identified during the development, refactoring core components of the library, and adding new features aimed at improving state management, establishing consistent and robust logging, and automating the testing

[1] https://github.com/Kristina-Pianykh/flink-multinode

process to monitor the system's overall behavior and ensure correctness of the matching mechanism.

5.1 Architecture

The system design involves nodes within a given topology communicating over a shared network, either as separate processes on the same host or as distinct hosts. Communication between nodes is facilitated through TCP sockets, and each node is equipped with a monitoring component that tracks event rates and continuously recomputes both sides of inequality 2 to react to significant fluctuations in event frequencies. Event generation is simulated by a stub service that reads from pre-generated event traces, consisting of event timestamps and IDs, which are then sent to the respective nodes. Each node has its own trace-generating stub running in a separate process. A detailed conceptualization of the system design can be found in Figure 7.

Figure 7: System architecture

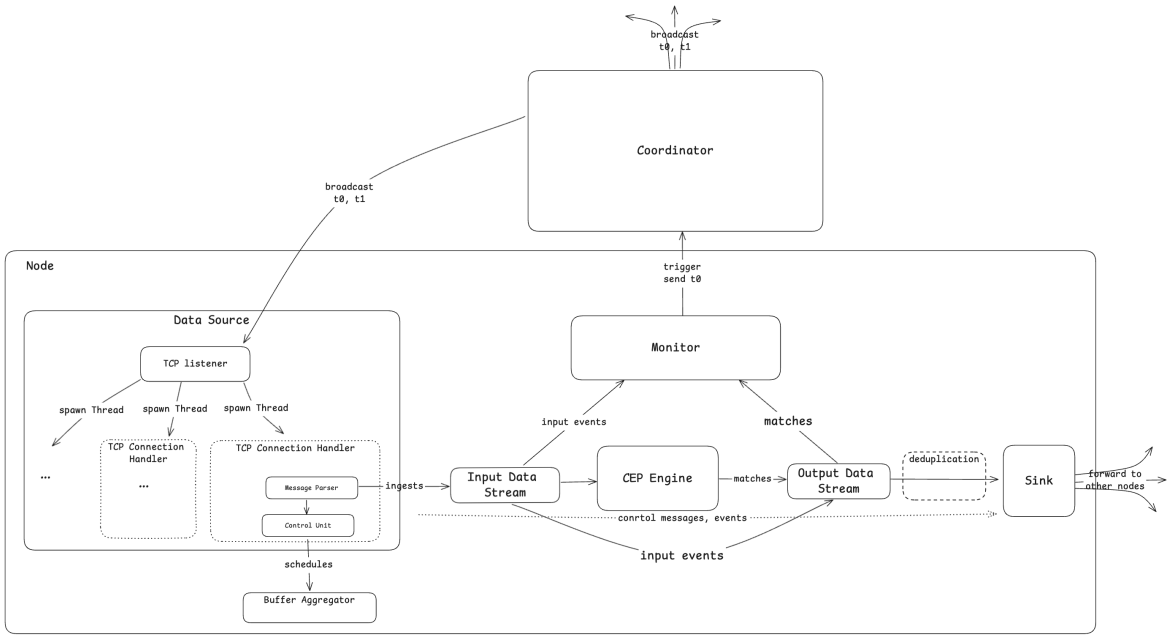


Figure 7 outlines the architecture of the key components in the distributed system with the focus on the `Node` and the `Coordinator` and demonstrates how they interact. The major components of the data processing pipeline and the steps involved in initiating the repair strategy are detailed below.

Data Source. The `Data Source` represents the entry point for incoming data on `Node`. It hosts a TCP server that listens for incoming connections. The communication on this end of the processing pipeline is one-way only since the outbound traffic is handled by the `Sink` so the writing end of the TCP data stream on opening a new socket is immediately closed.

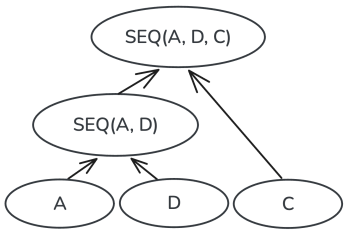
- `TCP Connection Handler`. Once the server detects an incoming connection, it

outsources its processing to a thread represented by the `TCP Connection Handler`, which makes it possible to parallelize handling multiple incoming connections.

- `Message Parser`. Performs message deserialization and parsing.
- `Control Unit`. If the data received is a control message, the `Control Unit` takes care of setting some state flags to remember the key timestamps (i.e. the timestamp $t0$ when a violation of the condition for the multi-node placement was detected and/or the timestamp $t1$ for the system state transition). Most importantly, on receiving a control message with the timestamp $t0$ the `Control Unit` schedules a `Buffer Aggregator` task described below.
- `Buffer Aggregator`. A task scheduled by the `Control Unit` to buffer events of the partitioning input up until the time $t1$ when the buffer is flushed to forward the aggregated events to the fallback node $v_{fallback}$. This task can only be initialized on the nodes V_{multi}.

Input Data Stream. All valid messages are ingested into the `Input Data Stream`, the entry point of the actual data processing pipeline that consists of the `Input Data Stream`, `CEP Engine`, and the `Output Data Stream`. The data in this stream is filtered into further streams for events and control messages to be handled separately. Control messages are ingested into the stream as well in order to propagate critical operational information to the components downstream, e.g. to the `Sink` which is also partly responsible for the management of the node's state (see the note on Flink's limitations in Section 7). The inputs $\lambda(p)$ for the multi-sink projection are sent to the `Monitor`.

Figure 8: Decomposition of $SEQ(A, D, C)$ with $\|SEQ(A, D, C)\| = 3$ for performance optimization in Flink CEP



CEP Engine. The core component responsible for query evaluation. Queries of a size bigger than 2 are decomposed to be evaluated in a series of chained sub-queries, each of size 2, resulting in a hierarchy that can be best conceptualized as a directed acyclic graph (DAG) in the form of a binary tree with bottom-up data flow where each node starting from the root has two children, or inputs. This decomposition does not alter the evaluation of the query workload and does not interfere with the originally generated INEv evaluation plan. Rather, this method serves the purpose of performance optimization by reducing the size of the internal state that Flink has to maintain. For instance, the query $q = SEQ(A, D, C)$ with the primitive inputs $\lambda(q) = \{A, C, D\}$ would be split into $q_1 = SEQ(A, D)$ with $\lambda(q_1) = \{A, D\}$ and $q_2 = q$ with $\lambda(q_2) = \{SEQ(A, D), C\}$, as

shown in Figure 8. Despite its resemblance to the notion of a projection in the context of INEv graphs, this query-splitting does not serve any purpose other than performance optimization. The output of the `CEP Engine` is a stream containing matches. The match events for the multi-sink projection are also sent to the `Monitor`.

Output Data Stream. The data stream that results from merging the `Input Data Stream` with the matches produced by the `CEP Engine`. In the final stage of the data processing pipeline, the elements in the `Output Data Stream` undergo the deduplication process to ensure that no event is released into the network more than once. This step is crucial to prevent duplicate matches, which could otherwise lead to incorrect event rates and compromise evaluation accuracy.

Algorithm 1 Rate Monitoring

Inputs: partitioning and non-partitioning inputs $p_{part}, \mathfrak{B}_{np} \in \lambda(p)$, respectively; p's matches; size of the Steiner tree $\|ST(f(i), G_{top})\|$ between the nodes $p' \in \mathfrak{P}_{np}$ supplying non-partitioning inputs

```
1  inequalityHolds := true
2  consecViolations := 0
3  windowSize := 10 sec
4  recomputeInterval := 1 sec
5  buffer, rates
6
7  function updateRates
8      for input in buffer do
9          rates.get(input) ← count(buffer.get(input))
10     end for
11 end function
12
13 function acceptConnection
14     socket ← initSocket()
15     event ← readEvent(socket)
16     if event.getTimestamp() ≤ now() and event.getTimestamp() ≥ (now() - windowSize) then
17         buffer.append(event)
18     end if
19 end function
20
21 function checkForDrift
22     if computeLHS(rates.get(p_part)) < computeRHS(‖ST(f(i), G_top)‖, rates.get(𝔓_np), rates.get(p_matches))
   then
23         if inequalityHolds == false then
24             consecViolations ← consecViolations + 1
25         else
26             inequalityHolds ← false
27         end if
28     else
29         inequalityHolds ← true
30     end if
31 end function
32
33 while True do
34     thread(acceptConnection())
35     checkForDrift()
36     if inequalityHolds == false and consecViolations ≥ 5 then
37         sendControl(now())
38     end if
39     Sleep(recomputeInterval)
40     updateRates()
41     clearOldEvents()
42 end while
```

Sink. The component that serializes events and forwards them to other nodes in the

network based on the rules defined in the node's forwarding table. In addition, the `Sink` shares the responsibility for managing the node's state by extracting instructions from control messages propagated down the data processing pipeline. This includes storing the updated forwarding table received from the `Coordinator` and performing the table swap at the specified timestamp.

Monitor. A critical component for initiating the repair process within the system. Similar to the `Data Source`, the `Monitor` hosts a TCP server that listens for the inputs of the projection p with a multi-node placement $\lambda(p) = \mathfrak{P}_{np} \cup \{p_{part}\}$, as well as p's matches. Its primary function is to continuously recompute both sides of inequality 2 at one-second intervals within a sliding window of 10 seconds. If the inequality fails to hold for five consecutive seconds — a threshold set to accommodate minor fluctuations in network and event frequencies — the `Monitor` notes the timestamp t_0 when the rate drift was confirmed and initiates the adaptive strategy by sending a control message with t_0 to the `Coordinator`.

Algorithm 2 Fallback Node Nomination

Inputs: V_{multi}; $\lambda(p)$; $\{\mathfrak{P}_i \subset \mathfrak{P} | i \in \|V_{multi}\| \text{ and } p \in \lambda(\mathfrak{P}_i)\}$
Output: $v_{fallback} \in V_{multi}$

```
1  function hasBiggestTable(V)
2      result := []
3      Vsorted = V.sortByFwdTableSize(asc=False)
4      maxTable := Vsorted.get(0)
5      for v in Vsorted do
6          if ||v.getFwdTable()|| == maxTable then
7              result.append(v)
8          end if
9      end for
10     if ||result|| > 1 then return resolve(result)
11     end if
12     return result.get(0)
13 end function
14
15 function determineFallbackNode(λ(p), V_multi)
16     result := []
17     for v_i in V_multi do
18         if procComplexChild(v_i, λ(p)) or procParent(v_i, 𝔓_i) then
19             result.append(v_i)
20         end if
21     end for
22
23     if ||result|| == 0 then
24         return hasBiggestTable(V_multi)
25     else if ||result|| > 1 then
26         return hasBiggestTable(result)
27     end if
28     return result.get(0)
29 end function
```

Algorithm 1 provides a high-level overview of the monitoring process. The server operates within a `While`-loop continuously listening for incoming connections, as shown in lines 33 to 42. When a connection is detected, the `acceptConnection()` function is called, spawning a new thread to handle the connection (line 34). The `acceptConnection()` function (lines 13-19) processes the incoming request by opening a socket and extracting the event from the payload. It checks the event's timestamp and adds it to the buffer

initialized on line 5 if the event was generated within a 10-second window from the current time.

The partitioning and non-partitioning input rates, along with the match rate, are stored in a global data structure (initialized on line 5) and are recomputed every second (line 41). The validity of the condition for multi-node placement is checked by invoking the function `checkForDrift()` on line 35. This function recalculates both sides of inequality 2, and if the inequality no longer holds, it updates the `inequalityHolds` flag and increments the `consecViolations` global variable, which tracks the number of consecutive violations. If the number of violations reaches 5 (line 36), the system confirms a rate drift and sends a control message to the coordinator, marking the timestamp $t0$. Finally, on line 41, the `clearOldEvents()` function is called to remove expired events from the buffer.

Algorithm 3 Compute New Forwarding Tables

Inputs: INEv Graph $I = (V, E)$; V_{multi}; $v_{fallback}$; topology nodes V_{top}; event universe $\mathcal{E}$; partitioning input p_{part}
Output: updated forwarding tables $\{t_i : (v_i, \mathcal{E}) \rightarrow V_{top} | v_i \in V_{top}, i \in \|V_{top}\|\}$

```
1  V_nonFallback ← V' = {v ∈ V_multi | v ≠ v_fallback}
2  criticalInevPairs ← {v × v_fallback | v ∈ pre(V_nonFallback) and v generates p_part}
3  criticalPaths ← computeCriticalPaths(criticalInevPairs)        ▷ pre computes the predecessors of a node
4  for node in V_top do
5      removeRules(node, criticalPaths)
6      addRules(node, criticalPaths)
7  end for
8  return {t_i : (v_i, E) → V_top | v_i ∈ V_top, i ∈ ||V_top||}
9
10 function computeCriticalPaths(pairs:={(v, v_fallback) | v ∈ pre(V_nonFallback)})
11     criticalPaths := []
12     for (src, dst) in pairs do
13         path ← computeShortestPath(src, dst)
14         criticalPaths.append(path)
15     end for
16     return criticalPaths
17 end function
18
19 function removeRules(node, criticalPaths)
20     newTable := []
21     fwdTable ← node.getFwdTable()
22     for hop in fwdTble do
23         if hop ∈ criticalPaths and not hop.contains(p_part) then        ▷ remove non-critical hops
24             newTable.append(hop)
25         end if
26     end for
27     node.setFwdTable(newTable)
28 end function
29
30 function addRules(node, criticalPath)
31     for path in criticalPaths do
32         if node in criticalPaths then
33             nextHop ← path.nextHop(node)
34             node.getFwdTable().append((node, p_part) → nextHop))    ▷ add rules for the partitioning input on
   critical paths
35         end if
36     end for
37 end function
```

Coordinator. The centralized component that manages the system's transition from the original state to the target state with a single-node placement type for a given multi-sink projection. It represents a wrapper around a TCP server listening for control

messages that contain the timestamp $t0$ indicating a rate drift that persisted within a 5-second interval. Upon receiving such a control message, the `Coordinator` computes an updated evaluation plan as outlined in Chapter 4 under 3. This includes selecting a fallback node $v_{fallback} \in V_{multi}$ (see Algorithm 2) and updating the forwarding tables for all nodes (see Algorithm 3). The control message containing both $t0$ and $t1$, as well as the new evaluation plan is broadcast to all the nodes in the network, which are responsible for storing the updated forwarding tables, setting respective flags to start buffering partitioning events, and preparing for the transition.

The selection of the fallback node is determined by both the local configuration of the nodes and their connectivity to the rest of the network, with priority given to the former. As shown in Algorithm 2, the function `determindeFallbackNode()` (lines 15-29) iterates over the nodes $v_i \in V_{multi}$ with $i \in \|V_{multi}\|$ and checks whether node v_i processes either a complex input to p (that is if there exists $p' \in \lambda(p)$ s.t. $p' = (O, \beta)$ and $o_c \in O$) or a projection p'' with $p \in \lambda(p'')$. If either condition is satisfied, v_i is added to the list of potential candidates for the fallback node. These conditions define processing locality, favoring nodes that handle high-intensity processing tasks—specifically, those that either evaluate and pass complex inputs to match p locally or pass matches of p to a higher-order projection without network traffic leaving the node. In most cases, this criterion is sufficient to identify a single node in the topology that has one or both of these properties.

However, if for all $p' = (O, \beta) \in \lambda(p)$, $o_c \notin O$ (i.e., the projection only involves primitive inputs, line 23), or if multiple nodes meet the processing locality condition (line 25), the function `hasBiggestTable()` is called to resolve the situation. This function (defined on lines 1-13) identifies nodes with the largest forwarding tables. If the result still yields multiple nodes with equally large tables, the `resolve()` function on line 10 randomly selects one of the candidates as the fallback node.

The final and most complex step in recomputing the evaluation plan for a given event network is updating the forwarding paths to accommodate the single-node placement type (see Algorithm 3). As discussed in the conceptual representation of the adaptive strategy in Section 4, this process involves incrementally updating the forwarding tables for each node in the topology, as represented by the `for`-loop on lines 3-7. The first step is to identify the critical INEv node pairs required for path recomputation and store them in the variable `criticalInevPairs`. The function `computeCriticalPaths()` (line 2) then calculates the shortest paths from each source node in `criticalInevPairs` that supplies the partitioning input p_{part}. Next, the function `removeRules()` (line 5) uses these critical paths to identify and remove the forwarding rules for non-critical paths on nodes supplying the partitioning input, and the result is stored on the node by updating its table object. The `addRules()` function (line 6) then reuses the previous result to add new rules that forward the partitioning input on the nodes along the critical paths (lines 32-35), which were computed earlier. The result is a set of (updated) forwarding tables for all $v \in V_{top}$ (line 8).

5.2 Flink CEP Library

Flink CEP is a library built on top of the open-source distributed data streaming framework Apache Flink Apache Software Foundation [2019a]. In the domain of distributed complex event processing (DCEP) systems, it is a widely recognized industry standard with such features as data processing parallelization, windowing operations, and a range of consumption policies for post-processing. These policies, ranging from the greedy *no-skip* to more granular options like *skip-to-first* or *skip-to-last*, help to reduce the number of final matches and manage memory load effectively. Flink CEP is implemented using a non-deterministic finite automaton (NFA) backed by a memory-optimized buffer to maintain all potential matches in active memory based on the paper Agrawal et al. [2008]. The limitations associated with Flink's state management are discussed in Section 7.

Despite some constraints in Flink's state management (detailed in Section 7), the library provides a functional programming API and extensive customization options for system initialization. In our implementation, we employed the *no-skip* greedy consumption policy to ensure the correct and complete pattern evaluation since the probabilistic semantics of a predicate check given a certain event is already expressed via the notion of selectivity.

6 Experiments

The optimization potential of the adaptive strategy was estimated based on the results of running a series of simulations of distributed in-network query evaluation in systems with and without the repair. The experiments were conducted several times, and the results were averaged to minimize the impact of variability caused by differences in query selectivities.

6.1 Topology Selection Criteria

In order to estimate the optimization potential of the adaptive strategy, the variety of topologies selected for running experiments was restricted to meet certain criteria.

1. The graphs with chained paths Concas et al. [2021] were not included in the test sample. A chained graph is a type of a graph $G = \{V, E\}$ where the nodes in V can be subdivided into 2 disjoint sets $V = V_1 \cup V_2$ s.t for $i \in \{1, 2\}$ $v_i \in V_i$ is only adjacent to some $v_j \in V_j$ for some $j \neq i$.

   Chained graphs represent a refinement of bipartite graphs [Concas et al., 2021, p. 2], where the vertices are sequentially connected. These topologies were excluded because they obscure the evaluation of the adaptive strategy's effectiveness. In such topologies, the forwarding paths remain largely unaffected by changes in query placement. For instance, if a multi-sink query is placed at nodes far apart from one another, partitioning events would still need to traverse long distances across the network, reducing the potential benefits of the adaptive strategy.

2. There can only be one multi-sink projection per event network. While it is possible to extend the repair strategy to networks with multiple multi-sink projections (see Section 7), the current study strives to eliminate potential noise factors that could arise from increasing the complexity of the implementation. Instead, the focus is put on investigating the optimization potential of the strategy before extending it to cover a wider range of network types.

3. The range of query operators is restricted to the minimal set of the $AND4$ and SEQ operators.

6.2 Experiment Design

The event networks generated for experiments fall into two groups: one group evaluates the query $q_1 = SEQ(A, B, C)$ and the other evaluates the query $q_2 = AND(A, B, C)$. Both queries have the fixed length $\|q_1\| = \|q_2\| = 3$ and evaluate events constructed from event universe $\mathcal{E} = \{A, B, C\}$. Each of these queries was evaluated in topologies ranging from 5 to 9 nodes. Each simulation was limited to the duration of 10 minutes.

To quantify the impact of the adaptive strategy, we need to simulate a scenario in which exchanging the non-partitioning inputs between the multi-sink nodes V_{multi} inflicts a higher TC as opposed to forwarding all of the multi-sink query inputs to one node. This can be achieved by artificially inflating the rates of the non-partitioning inputs, forcing the system to initiate the repair protocol. This consideration had a significant impact on the choice of the event networks for the test sample. Specifically, the preference was given to generating networks for which the constructed INEv graph did not result in a decomposition of the original query into projections involving complex operators. In other words, the evaluated queries were required to accept only primitive event inputs. The reason for this is that manipulating the rates of inputs represented by primitive events has a more direct and measurable impact on the query's match rate, which allows to modify input rates with a controlled outcome. The example below demonstrates why:

Example 6.1. Consider the query $q' = SEQ(A, B, C)$ with $\lambda(q') = \{A, B, C\}$ primitive inputs. Consider the query $q'' = SEQ(A, B, C)$ with $\lambda(q'') = \{p, C\}$ where $p = SEQ(A, B)$ a projection. Based on the definition for the match rate of a complex operator from 2.2, the match rates for q' and q'' are computed as follows:

$r(q') = \sigma(q') \cdot \Pi_{1 \leq i \leq k} r(\lambda(q')) = \sigma(q') \cdot r(A) \cdot r(B) \cdot r(C)$
$r(q'') = \sigma(q'') \cdot \Pi_{1 \leq i \leq k} r(\lambda(q'')) = \sigma(q'') \cdot r(p) \cdot r(C) = \sigma(q'') \cdot (\sigma(p) \cdot \Pi_{1 \leq i \leq j} r(\lambda(p))) \cdot r(C) =$
$\sigma(q'') \cdot \sigma(p) \cdot r(A) \cdot r(B) \cdot r(C)$

With $\sigma(q') = \sigma(q'')$ and $r_{q'}(\epsilon) = r_{q''}(\epsilon)$ for each $\epsilon \in \mathcal{E}$, $r(q') > r(q'')$ because $\sigma(o)$ for any operator ranges from 0 to 1.

Example 6.1 shows how decomposing a query into arbitrary trees of a greater depth impacts the resulting match rate. Each additional level of depth contributes incrementally

to the final result at the root level. As a result, the impact of altering the rates of primitive events, represented as the leaves in the query tree, diminishes with each additional level of hierarchy. Consequently, the influence of manipulating the rates of primitive query inputs becomes increasingly negligible. The deeper the query tree, the more challenging it becomes to estimate the potential optimization of the repair strategy. For this reason, the majority of the selected event networks (9 out of 11) evaluate the queries q_1 and q_2 with $\lambda(q_1) = \lambda(q_2) = \{A, B, C\}$ primitive input types. Having said that, two of the selected networks evaluate the queries with $\lambda(q_1) = \lambda(q_2) = \{AND(A, B), C\}$ complex inputs to investigate the effects of adding an additional level of hierarchy in the query tree, similar to q'' in Example 6.1.

To systematically inflate input rates across various event networks in order to cause the system to trigger a repair, we employed a structured approach by adjusting the event node ratios, i.e., the proportion of event types generated per node. The details of this method are outlined in Algorithm 4. This scaling approach normalizes node event ratios while introducing systematic rate inflation and maintaining event skew. The goal is to preserve the inherent relationship between event types while artificially inflating their rates. The system of gradually increasing scaling factors allows us to regulate the scaling of event rates and test the effectiveness of the repair strategy in a controlled manner.

Algorithm 4 Scaling Rates Algorithm

Inputs: node event ratios originalRatios
Output: a matrix with event node ratios where non-partitioning input rates are inflated by a factor

```
1  function SCALERATIOS(originalRatios)
2      ratioMatrix = []
3      ratios_np = []
4      ratio_p ← MAX(originalRatios)                          ▷ get the ratio of the partitioning inputs
5      for ratio in originalRatios do
6          if ratio ≠ ratio_p then ratios_np.APPEND(ratio)    ▷ get the ratios of the non-partitioning inputs
7          end if
8      end for
9      for α in [0.2, 0.3, 0.5, 0.7, 1.0, 1.3, 1.5, 2.0] do
10         newRatios = []
11         scalingFactor ← ratios_np / (ratio_p · α)
12         for ratio in originalRatios do
13             if ratio ≠ ratio_p then
14                 newRatios.APPEND(ratio · scalingFactor)
15             else
16                 newRatios.APPEND(ratio)
17             end if
18         end for
19         ratioMatrix.APPEND(newRatios)
20     end for
21     return ratioMatrix
22 end function
```

The rates for the non-partitioning inputs were increased starting from the 3rd minute from the beginning of a simulation.

6.3 Results

The results of applying the adaptive strategy in networks of various sizes and evaluation plans for the queries q_1 and q_2 are summarized in Figure 9 and Figure 10, respectively. These figures depict transmission costs (TC) measured in the total number of events disseminated in the network, both with and without repair, under varying inflation factors for increasing the rates of the non-partitioning inputs. For clarity, the data is normalized against the baseline network costs observed in the simulations where the adaptive strategy was not applied, centering these baseline values around 1.0. This normalization facilitates a clear comparison across different topologies and highlights the impact of the repair strategy relative to the baseline. The exact values for the TC obtained in each simulation run for every individual topology, as well as the quantified impact of the adaptive strategy can be found in Table 2 and Table 3 in Appendix 8.1.

Figure 9: TC in total number of events sent with repair (orange) and without repair (blue) for topologies with $Q = SEQ(A, B, C)$

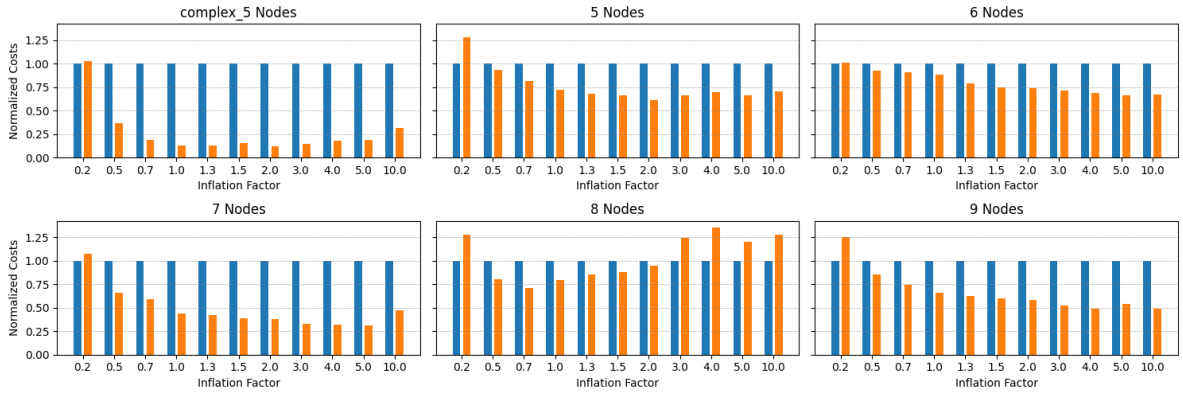
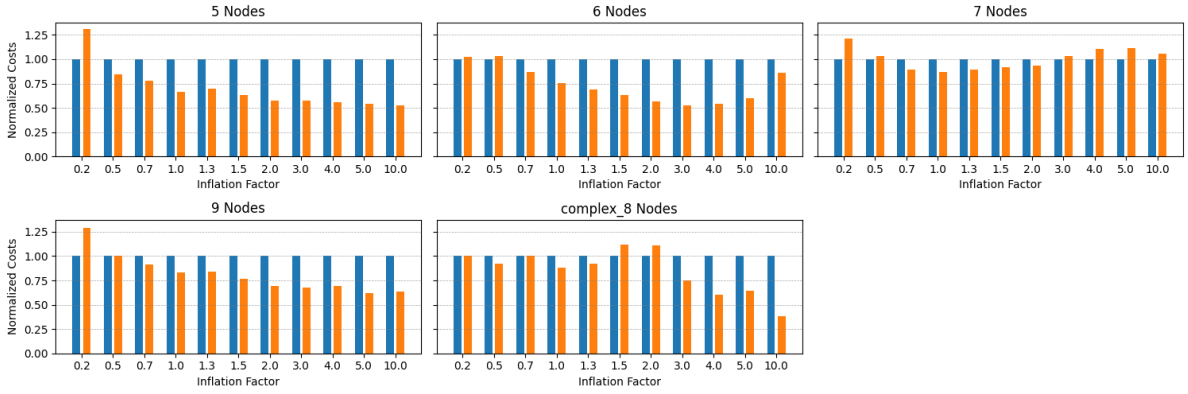


Figure 10: TC in total number of events sent with repair (orange) and without repair (blue) for topologies with $Q = AND(A, B, C)$



The general data pattern suggests the idea that the optimization potential of the repair strategy is roughly aligned with the magnitude of an increase in rates among events of the non-partitioning input type. Initiating the repair in the experiments where the rates of the non-partitioning inputs were increased just marginally (by the factor of 0.2 and

0.5 of the rate of the partitioning input), not only did not result in a lower amount of data in the network but it even caused a slightly higher TC compared to the baseline. This might be attributed to the insufficient difference between the rate ratios of the partitioning and non-partitioning inputs, especially when contrasted with the impact of the repair strategy as the inflation factor grows bigger. The reduction of the amount of data disseminated in the network reaches up to 62.2% in the group of event networks evaluating $AND(A, B, C)$ and up to 87.8% in the group evaluating $SEQ(A, B, C)$. On average, switching from a multi-node to a single-node query placement when the former setup becomes harmful leads to a 29% decrease in a TC (note that the value for cost savings in the experiments where the repair did not bring any benefits was set to 0).

The shape of the data in Figure 9 and Figure 10 suggest that there is no apparent correlation between the size of an event network and the effectiveness of the repair: the benefits of the adaptive strategy are most apparent in the topologies with $\|V_{top}\| \in \{5, 7, 9\}$ in the group evaluating $SEQ(A, B, C)$ and $\|V_{top}\| \in \{5, 6, 9\}$ in the group evaluating $AND(A, B, C)$.

Variation of TC by Inflation Factor

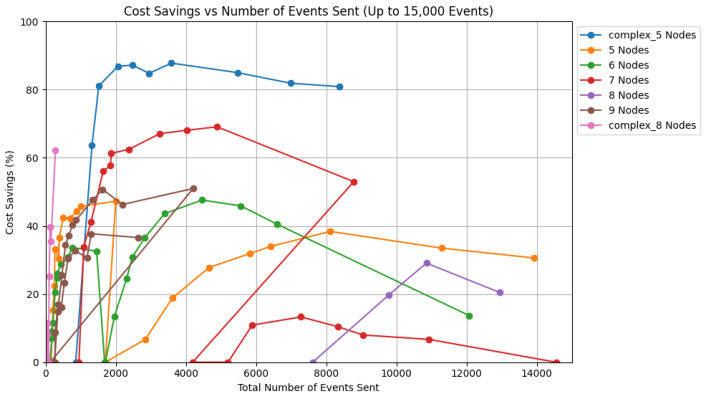
A peculiar observation can be made about the U-shape of the TC when the repair was enabled in nearly all of the simulations. Specifically, for small increases in the rates of non-partitioning inputs, the repair of the query placement type does not yield substantial data transmission savings. As the rates of non-partitioning inputs increase, the simulations with the repair strategy enabled exhibit progressively greater benefits. Interestingly, when the inflation factor approaches its maximum, at $\alpha = 10.0$, the TC begins to rise again, and in some cases, even surpasses the baseline values. At first glance, this may seem counter-intuitive, as one would expect the difference in TC with and without the repair to increase as the non-partitioning input frequencies grow.

A more detailed analysis reveals that the variation in event node ratios across the generated event networks correlates directly with the amount of data transmitted. Higher event node ratios result in a larger amount of data being transmitted within the network, which in turn causes an overload of the Flink CEP engine, making it the primary throughput bottleneck. For instance, the total number of events disseminated in the network evaluating $SEQ(A, B, C)$ and the topology $\|V_{top}\| = 8$ exceeds 70000 with the strategy enabled and 45000 without it (the exact values can be found in Appendix 8.1). Similarly, in the network evaluating $AND(A, B, C)$ and the topology $\|V_{top}\| = 7$, the transmitted data amounts to nearly 30000 elements. In contrast, the network evaluating $SEQ(A, B, C)$ with $\|V_{top}\| = 6$ transmits merely up to 2000 events throughout the entire simulation, resulting in a clear and expected pattern for TC savings. In this case, the TC decreases gradually as the inflation factor for manipulating the rates of the non-partitioning inputs increases, which aligns with our initial expectations and hypothesis regarding the optimization potential of the adaptive strategy.

Engine Overload and Latency

At first glance, one might expect a direct correlation between the cost savings achieved through the repair strategy and the total number of forwarded events. However, the data reveals a more intricate relationship. The patterns observed suggest that other factors are at play, which are not immediately apparent when considering cost savings alone.

Figure 11: Cost saving as a function of the number of transmitted events in the network

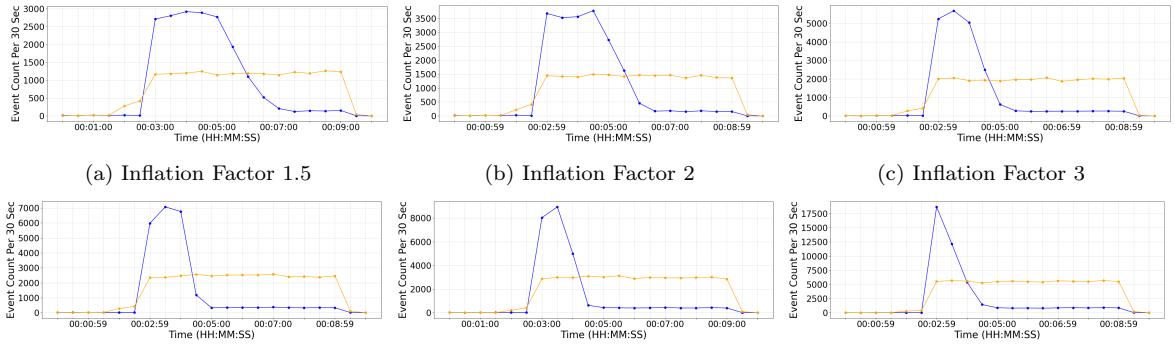


As illustrated in Figure 11, the optimization potential of the adaptive strategy that is especially prominent in the first half of the simulation period appears to lose its momentum as the network becomes saturated, which occurs when the cumulative amount of transmitted data exceeds the threshold of roughly 10000 events within a 10-minute simulation period. A closer inspection of the event rate distributions across different experiments (see Appendix 8.2), however, reveals a sudden drop in the number of transmitted events in the simulation runs without the repair once the total input rate has reached a certain level. This behavior is especially prominent in the event networks where the amount of transmitted data exceeds 2500 events within a 30-second interval, i.e. the total event rate surpasses approximately 83 events/s (as shown in Figure 15, Figure 17, Figure 18, Figure 21, Figure 22).

If we take a look specifically at the data patterns of the event network evaluating $SEQ(A, B, C)$ with $\|V_{top}\| = 8$ and the highest event node ratios compared to those in the other event networks in our test sample, we can observe that once the data transmission frequency spikes immediately after the non-partitioning input rates are increased by the inflation factor $\alpha > 1.5$ at the 3rd minute after the start of the simulation, it plummets as fast and remains on that level until the end of the run (Figure 12. The bigger the inflation factor for increasing the non-partitioning input rates, the faster it leads to a halt when no more data is emitted into the network. This is due to the limited capacity of the Flink CEP engine to process data with increasingly growing throughput. The internal state reaches a considerable enough size to immediately become a bottleneck for the entire data processing pipeline. This provides an explanation for the greater TC incurred in the simulation runs with the adaptive strategy enabled as opposed to the baseline

values where no more data was emitted due to being blocked by the computationally intensive workload in the CEP engine.

Figure 12: Variation of TC by inflation factor for increasing the non-partitioning inputs in the even network evaluation $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 8$



(a) Inflation Factor 1.5 (b) Inflation Factor 2 (c) Inflation Factor 3

(d) Inflation Factor 4 (e) Inflation Factor 5 (f) Inflation Factor 10

The assumption about the throughput constraints within the engine is supported by the analysis of the processing latency, i.e. the amount of time elapsed from the moment an event resulting in a match is supplied to the Flink CEP engine until the moment when a match is produced. As shown in Figure 13, the data demonstrates a clear linear relationship between the input frequency and the processing latency. Due to the complexity of the state management within Flink's engine and the combinatorial nature of query construction leading to the exponential growth in the state size in worst-case scenarios as reported in Ziehn [2020], data processing is significantly throttled. Similarly, the issues related to handling high throughput with the Flink CEP library are also reported in Giatrakos et al. [2021].

7 Limitations and Future Work

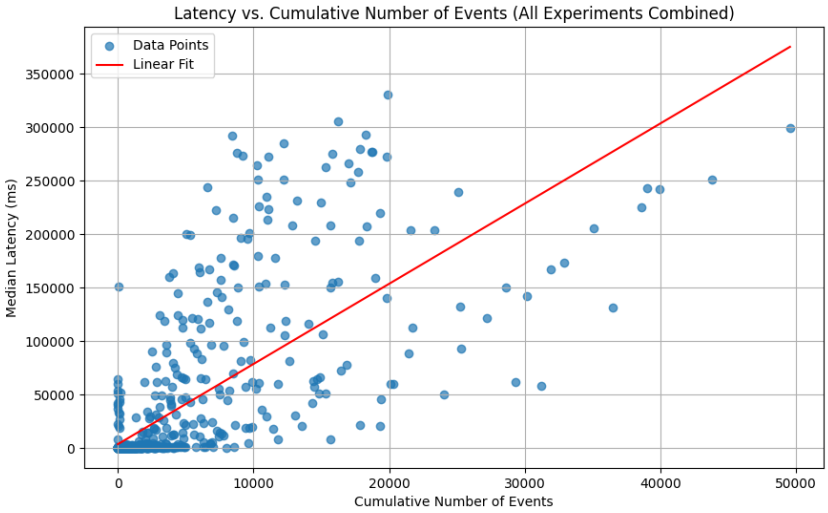
The adaptive strategy presented in this study has been scoped to analyze its optimization potential given networks of a relatively small size and query workload. We, therefore, propose the following ideas to explore its impact in more complex contexts.

Event node ratios as influence factor. One of our findings revealed that Flink CEP struggles with high-frequency inputs, leading to bottlenecks and significant processing latency. This issue became apparent after the experiments were designed and executed. Future work should examine the impact of the repair strategy while controlling event node ratios to keep the amount of transmitted data low, thus minimizing the influence of Flink's limitations on the effects of the strategy.

Reverting the repair. Implementing a feature that allows dynamic switching between single-node and multi-node placements as rates fluctuate could greatly improve the system's flexibility.

Wider range of query operators. The effects of the adaptive strategy are investigated in the current study based on the queries with a limited number of operators

Figure 13: The relationship between the cumulative number of events in the network at a given minute and processing latency



with $O = \{SEQ, AND\}$. Expanding this set to a wider variety of operators, e.g. Kleene closure, negation, and disjunction (as represented by OR), can explore other aspects of the performance of the strategy and be more suitable to address real-world scenarios involving complex evaluation patterns.

Multiple multi-sink projections. The current conceptualization of the repair strategy does not yet address scenarios involving multiple multi-sink projections. For instance, in a query such as $q = SEQ(A, B, C)$ with $\lambda(q) = \{p, C\}$ where $p = SEQ(A, B)$ s.t. for the corresponding INEv graph $I = (V, E)$, $\|\{v \in V | v = (p, v_{node}, v_{graph}, v_{os}\}\| > 1$ and, similarly for p_q, $\|\{v \in V | v = (p_q, v_{node}, v_{graph}, v_{os}\}\| > 1$, where p_q is the top-level projection in the INEv graph corresponding to q, p and p_q are multi-sink projections. Let the event rates be $r(A) \ll r(B) \ll r(C)$. While a significant increase in A's rates will most certainly trigger a fallback to a single-node placement of p, the impact of this change on transmission costs in combination with the multi-sink placement of p_q remains unclear. Another challenge arises from a scenario involving an increase in $r(B)$, as it becomes harder to determine the optimal conditions for triggering the repair, particularly in scenarios involving interdependent multi-sink projections. Moreover, constructing a new evaluation plan involving multiple multi-sink projections might be more computationally intensive which can come at the detriment of the reaction time in a real-time application.

Generic strategy. Given previous research on developing an adaptive strategy to repair invalid projections (Palm [2023]), it might be interesting to combine it with the strategy designed in the current study to produce a more generalized approach in order to facilitate a flexible recovery in contexts when fluctuations in rates cause higher transmission costs.

Acknowledgment signaling for reliability. Implementing a protocol that requires nodes to acknowledge readiness before initiating the repair could make the adaptive

strategy more transactional by introducing a guarantee of atomicity. Additionally, it may enable timely detection and handling of overload situations. However, this approach risks leading to deadlocks if processing is significantly hindered and the coordinator continues waiting for acknowledgment from an overloaded node. The challenge remains in developing a strategy to resolve this issue.

Distributed coordination. Embedding the coordinator within one of the nodes in the topology would remove the centralized component from the distributed system. This, however, would require nodes to be equipped with network-wide knowledge to accurately forward control messages.

Flink-related challenges

No support for dynamic parametrization. Flink lacks support for runtime parametrization, as it generates an immutable execution plan during initialization. Similarly, updating, enabling, or disabling query patterns within the CEP engine is not supported, which proved to be a significant challenge in applying runtime modifications required by the repair strategy. To bypass this limitation, instructions were wrapped into control messages and ingested into the input data stream, allowing downstream components to extract and apply updates locally.

Challenges in state sharing. Flink supports state sharing between two streams, but synchronization through watermarking is problematic. Sharing state across more than two streams is not feasible, as it conflicts with Flink's design for distributed workloads, where data is spread across separate nodes. This limitation made certain implementations more challenging. For instance, instead of using Flink's native data structures for metrics, which would offer built-in monitoring, we had to develop a separate monitoring service.

8 Conclusion

The current study addressed the challenge of adaptivity in distributed event processing with in-network evaluation, with the goal of reducing transmission costs incurred by multi-sink queries when their placement becomes inefficient. The implemented adaptive strategy dynamically transitions the system from a multi-node to a single-node placement type for a given multi-sink query when fluctuations in the rates of the query's inputs incur higher transmission costs. The strategy employs several methods such as event buffering and retrospective matching to ensure completeness and correctness of the evaluation process and prevent data loss during the transition. Overall, our experimental results showed that this approach effectively mitigates undesirable scenarios by reducing the transmission cost increases caused by event rate fluctuations, compared to scenarios without the repair. By allowing dynamic reconfiguration of query placements, we also addressed the issue of maintaining system performance throughout the transition. The optimization potential of the repair strategy can be further explored by extending it to a wider range of contexts. Lastly, combining this strategy with methods developed in previous research could lead to more generalized solutions for adaptivity in distributed

event processing systems.

While the Flink framework provided a solid foundation for the implementation, its lack of runtime flexibility and the immutable nature of its execution plans posed certain challenges, especially when handling high-frequency data, which resulted in processing bottlenecks. The latter impacted the results of the experiments and interfered with the evaluation of the optimization potential of the strategy. Exploring alternatives to Flink that support the dynamic configuration of execution plans and handle high throughput could further optimize the system's adaptability and performance.

References

J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, page 147–160, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376634. URL https://doi.org/10.1145/1376616.1376634.

S. Akili, S. Purtzel, and D. Pahl. sigmod24-flink. https://cfgit.ddnss.de/cavan/sigmod24-flink, 2023a. Private repository, accessible only by invitation.

S. Akili, S. Purtzel, and M. Weidlich. Inev: In-network evaluation for event stream processing. Proc. ACM Manag. Data, 1(1), may 2023b. doi: 10.1145/3588955. URL https://doi.org/10.1145/3588955.

Apache Flink. Apache flink 1.14 documentation: Java api, 2021. URL https://nightlies.apache.org/flink/flink-docs-release-1.14/api/java/. Accessed: 2024-09-04.

Apache Software Foundation. Apache flink: Stateful computations over data streams, 2019a. URL https://flink.apache.org/. Accessed: 2024-09-07.

Apache Software Foundation. Apache storm: Distributed real-time computation system, 2019b. URL https://storm.apache.org/. Accessed: 2024-09-07.

A. Concas, L. Reichel, G. Rodriguez, and Y. Zhang. Chained graphs and some applications. Applied Network Science, 6(1):39, 2021. ISSN 2364-8228. doi: 10.1007/s41109-021-00377-4. URL https://doi.org/10.1007/s41109-021-00377-4.

G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. ACM Comput. Surv., 44(3), jun 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677. URL https://doi.org/10.1145/2187671.2187677.

I. Flouris, N. Giatrakos, A. Deligiannakis, and M. Garofalakis. Network-wide complex event processing over geographically distributed data sources. Information Systems, 88:101442, 2020. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2019.101442. URL https://www.sciencedirect.com/science/article/pii/S0306437919304946.

N. Giatrakos, E. Kougioumtzi, A. Kontaxakis, A. Deligiannakis, and Y. Kotidis. Easyflinkcep: Big event data analytics for everyone. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management, CIKM '21, page 3029–3033, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384469. doi: 10.1145/3459637.3482094. URL https://doi.org/10.1145/3459637.3482094.

E. Inc. Esper complex event processing (cep), 2024. URL http://www.espertech.com/esper/. Accessed: 2024-09-07.

M. Lima, R. Lima, F. Lins, and M. Bonfim. Beholder–a cep-based intrusion detection and prevention system for iot environments. Computers & Security, 120:102824, 2022. ISSN 0167-4048. doi: 10.1016/j.cose.2022.102824. URL https://www.sciencedirect.com/science/article/pii/S0167404822002188.

L. Palm. Operator replacement for adaptive distributed stream processing. Bachelor's thesis, Humboldt-Universität zu Berlin, 2023.

F. Paraiso, G. Hermosillo, R. Rouvoy, P. Merle, and L. Seinturier. A middleware platform to federate complex event processing. In 2012 IEEE 16th International Enterprise Distributed Object Computing Conference, pages 113–122, 2012. doi: 10.1109/EDOC.2012.22.

J. Roldán, J. Boubeta-Puig, J. L. Martínez, and G. Ortiz. Integrating complex event processing and machine learning: An intelligent architecture for detecting iot security attacks. Expert Systems with Applications, 149:113251, 2020. ISSN 0957-4174. doi: 10.1016/j.eswa.2020.113251. URL https://www.sciencedirect.com/science/article/pii/S0957417420300762.

N. Shah, S. Shah, P. Jain, and N. Doshi. Overview of present-day iot data processing technologies. Procedia Computer Science, 210:277–282, 2022. ISSN 1877-0509. doi: 10.1016/j.procs.2022.10.150. URL https://www.sciencedirect.com/science/article/pii/S1877050922016076.

K. Teymourian, M. Rohde, and A. Paschke. Knowledge-based processing of complex stock market events. In Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, page 594–597, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307901. doi: 10.1145/2247596.2247674. URL https://doi.org/10.1145/2247596.2247674.

A. Ziehn. Complex event processing for the internet of things, 08 2020.

X.-R. Zu, Y. Bai, and S.-L. Ma. Complex event processing for smart grid active management in distributed new energy generation environment. 01 2017. doi: 10.2991/eesed-16.2017.96.

Appendix

8.1 Tables

The transmission costs (TC) accumulated in the networks with a number of nodes ranging from 5 to 9, both with and without applying the repair strategy, are listed below in Table 1 for the query workload with a multi-node placement $q_1 = SEQ(A, B, C)$ and in Table 2 for $q_2 = AND(A, B, C)$. Each row represents the total number of events sent over the network with and without the repair strategy given a topology of a certain size (number of nodes in the network). The inflation factor refers to the ratio by which the rates of the non-partitioning inputs were increased relative to the rate of the partitioning input. The last column quantifies the decrease in data transmitted across the network when using the adaptive strategy. The values showing the greatest reduction in data transmission due to the repair strategy are highlighted in bold. Empty cells indicate that the adaptive strategy was not beneficial for decreasing TC.

Table 2: TC in the even networks evaluating $SEQ(A, B, C)$ with and without the repair strategy

Network Size	**Inflation Factor**	**Costs with Repair**	**Costs without Repair**	**Cost Savings**
5	0.2	1321	1697	-
	0.5	3044	2840	6.7%
	0.7	4444	3608	18.8%
	1.0	6448	4658	27.8%
	1.3	8544	5817	32%
	1.5	9726	6418	34%
	2.0	13179	8114	**38.4%**
	3.0	16978	11287	33.5
	4.0	20067	13931	30.6%
	5.0	23776	15694	34.0%
	10.0	39599	27905	29.5%
5_complex	0.2	828	854	-
	0.5	3633	1318	.63.7%
	0.7	7996	1514	81.1%
	1.0	15702	2062	86.9%
	1.3	19406	2475	87.2%
	1.5	19354	2947	84.8%
	2.0	29332	3583	**87.8%**
	3.0	36489	5473	85.0%
	4.0	38584	6979	81.9%
	5.0	43813	8375	80.9%
	10.0	49548	15548	68.6%
6	0.2	123	124	-
	0.5	185	172	7%
	0.7	177	161	9.1%
	1.0	234	207	11.5%
	1.3	336	267	20.5%
	1.5	428	322	24.8%
	2.0	444	328	26.1%
	3.0	614	438	28.7%
	4.0	934	646	30.8%
	5.0	1139	755	**33.7%**
	10.0	2153	1451	32.6%
7	0.2	880	947	-
	0.5	1638	1083	33.9%
	0.7	2189	1290	41.1%
	1.0	3734	1639	56.1%

Network Size	Inflation Factor	Costs with Repair	Costs without Repair	Cost Savings
	1.3	4358	1845	57.7%
	1.5	4838	1870	61.3%
	2.0	6327	2373	62.5%
	3.0	9895	3253	67.1%
	4.0	12616	4026	**68.1%**
	5.0	15829	4886	69.1%
	10.0	18702	8782	53.0%
8	0.2	4979	7610	-
	0.5	12195	9786	19.8%
	0.7	15298	10852	**29.1%**
	1.0	16263	12936	21.5%
	1.3	17516	15028	14.2%
	1.5	18522	16371	11.6%
	2.0	20464	19313	5.6%
	3.0	21250	26425	-
	4.0	24216	32838	-
	5.0	26444	39397	-
	10.0	45510	72903	-
9	0.2	203	254	-
	0.5	403	343	14.9%
	0.7	619	461	74.5%
	1.0	860	564	34.4%
	1.3	1050	659	37.2%
	1.5	1280	764	40.3%
	2.0	1478	861	41.7%
	3.0	2557	1340	47.6%
	4.0	3295	1624	50.7%
	5.0	4077	2189	46.3%
	10.0	8577	4203	**51.0%**

Table 3: TC in the even networks evaluating $AND(A, B, C)$ with and without the repair strategy

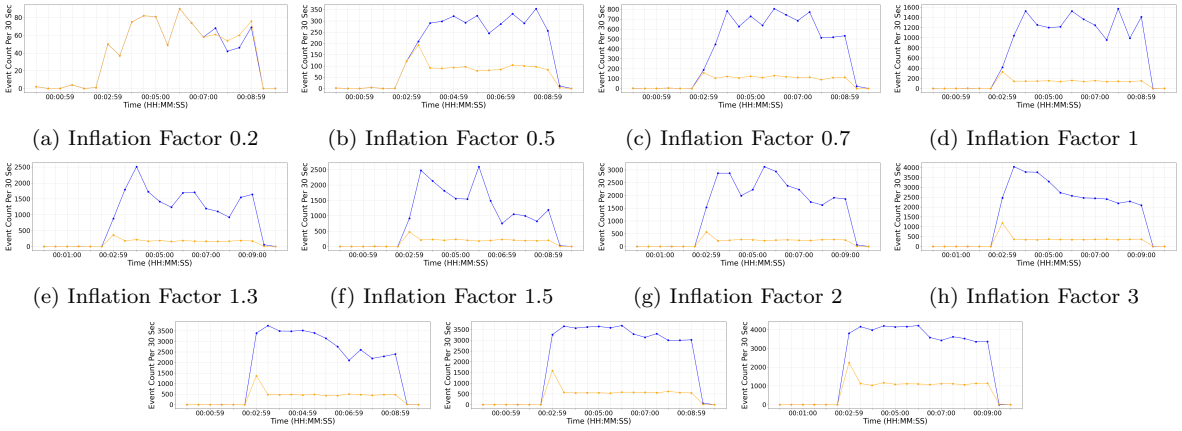
Network Size	Inflation Factor	Costs with Repair	Costs without Repair	Cost Savings
5	0.2	117	153	-
	0.5	241	204	15.4%
	0.7	321	249	22.4%
	1.0	394	263	33.2%
	1.3	533	371	30.4%
	1.5	607	385	36.6%
	2.0	870	501	42.4%
	3.0	1210	698	42.3%
	4.0	1589	884	44.4%
	5.0	1856	1008	45.7%
	10.0	3794	1998	**47.3%**
6	0.2	1643	1680	-
	0.5	1655	1702	-
	0.7	2253	1954	13.3%
	1.0	3060	2308	24.6%
	1.3	3565	2466	30.8%
	1.5	4438	2820	36.5%
	2.0	6005	3385	43.6%
	3.0	8502	4453	**47.6%**
	4.0	10273	5559	45.9%
	5.0	11081	6598	40.5%
	10.0	13987	12072	13.7%
7	0.2	2901	4190	-
	0.5	5023	5196	-
	0.7	6609	5886	11%
	1.0	8393	7278	**13.3%**
	1.3	9308	8336	10.5%

Network Size	Inflation Factor	Costs with Repair	Costs without Repair	Cost Savings
	1.5	9827	9045	8%
	2.0	11713	10931	6.7%
	3.0	14114	14552	-
	4.0	16149	17874	-
	5.0	17768	19744	-
	10.0	27652	29157	-
8_complex	0.2	29	29	-
	0.5	37	34	8.1%
	0.7	36	36	-
	1.0	43	34	20.1%
	1.3	49	45	8.2%
	1.5	61	64	-
	2.0	75	83	-
	3.0	131	98	25.2%
	4.0	206	124	39.8%
	5.0	231	149	35.5%
	10.0	732	277	**62.2%**
9	0.2	66	131	-
	0.5	262	263	-
	0.7	295	269	8.2%
	1.0	424	352	17%
	1.3	528	443	16.1%
	1.5	689	529	23.3%
	2.0	916	637	30.5%
	3.0	1245	838	32.7%
	4.0	1699	1178	30.7%
	5.0	2083	1298	**36.7%**
	10.0	4151	2630	36.6%

8.2 Plots

Variation in TC by the inflation factor for increasing the non-partitioning input rates with the repair strategy enabled (orange) and without it (blue) for each event network. The y-axis indicates the number of transmitted events, measured in 30-second intervals, while the x-axis represents the elapsed time, offset from 00:00:00, in HH:MM:SS format.

Figure 14: $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 5$ and $AND(A, B) \in \lambda(q_1)$

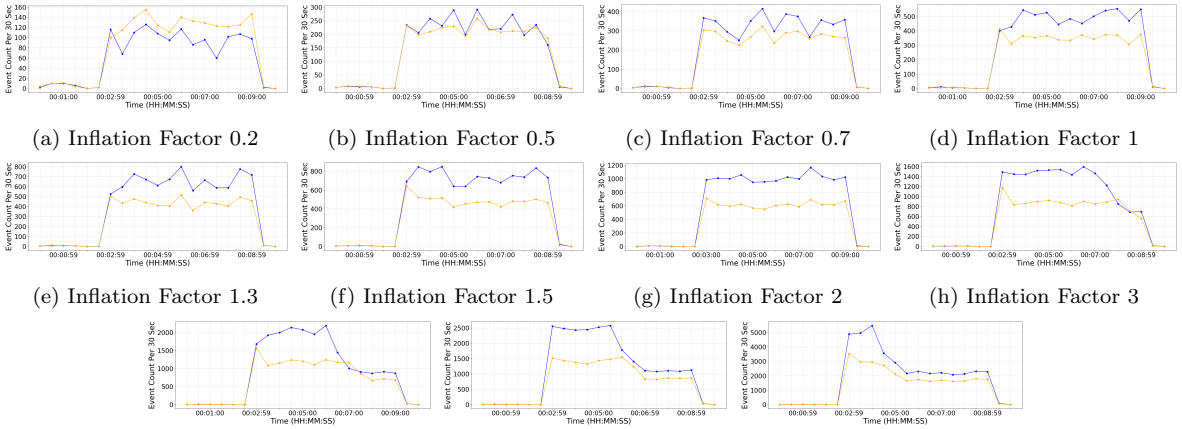


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 15: $SEQ(A, B, C)$ with $\|V_{top}\| = 5$

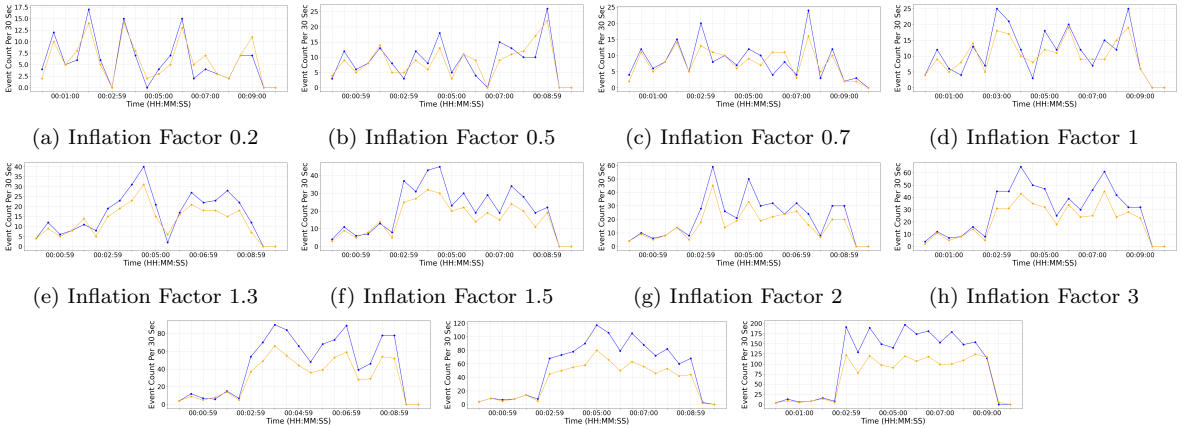


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 16: $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 6$

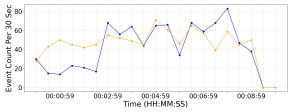


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

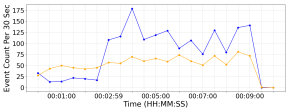
(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

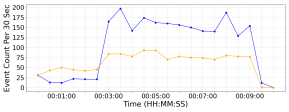
Figure 17: $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 7$



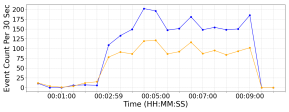
(a) Inflation Factor 0.2



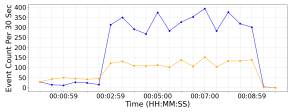
(b) Inflation Factor 0.5



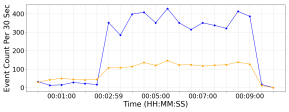
(c) Inflation Factor 0.7



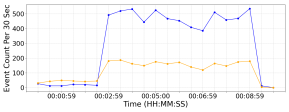
(d) Inflation Factor 1



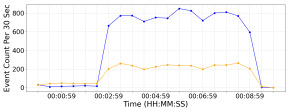
(e) Inflation Factor 1.3



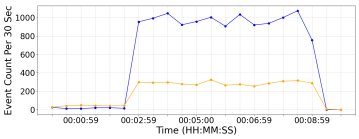
(f) Inflation Factor 1.5



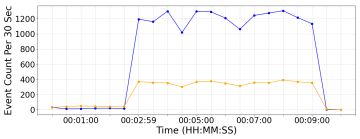
(g) Inflation Factor 2



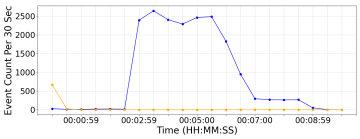
(h) Inflation Factor 3



(i) Inflation Factor 4

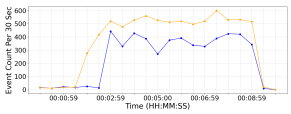


(j) Inflation Factor 5

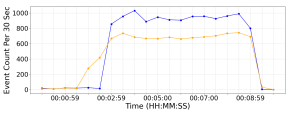


(k) Inflation Factor 10

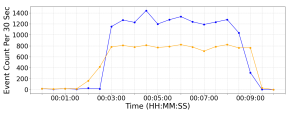
Figure 18: $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 8$



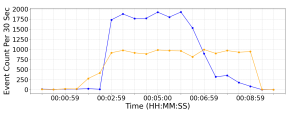
(a) Inflation Factor 0.2



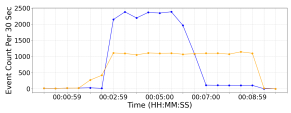
(b) Inflation Factor 0.5



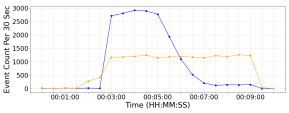
(c) Inflation Factor 0.7



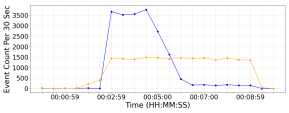
(d) Inflation Factor 1



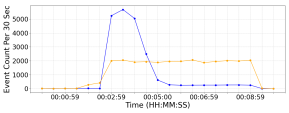
(e) Inflation Factor 1.3



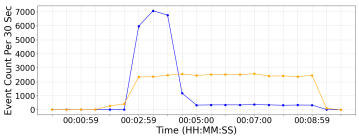
(f) Inflation Factor 1.5



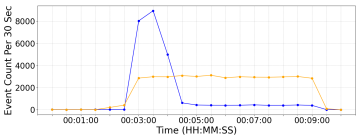
(g) Inflation Factor 2



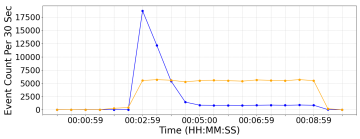
(h) Inflation Factor 3



(i) Inflation Factor 4

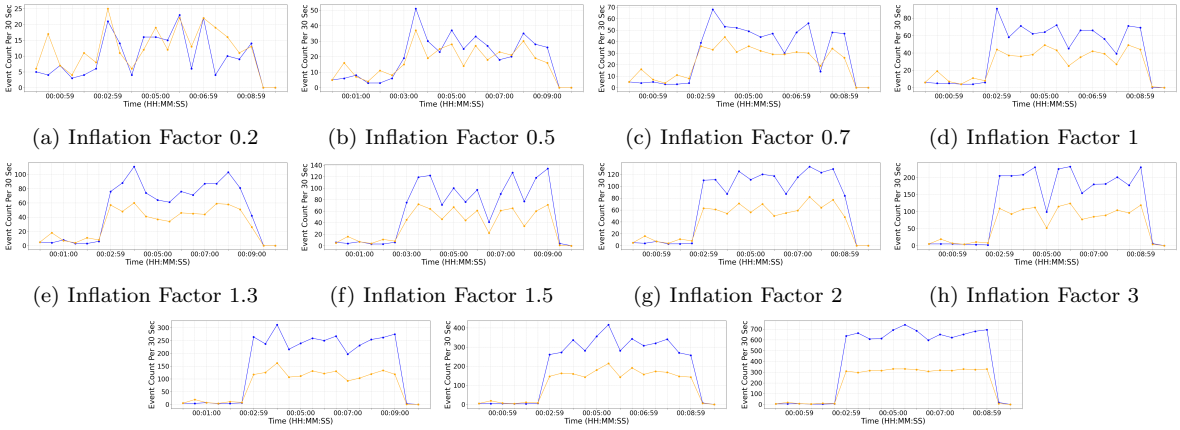


(j) Inflation Factor 5



(k) Inflation Factor 10

Figure 19: $q_1 = SEQ(A, B, C)$ with $\|V_{top}\| = 9$

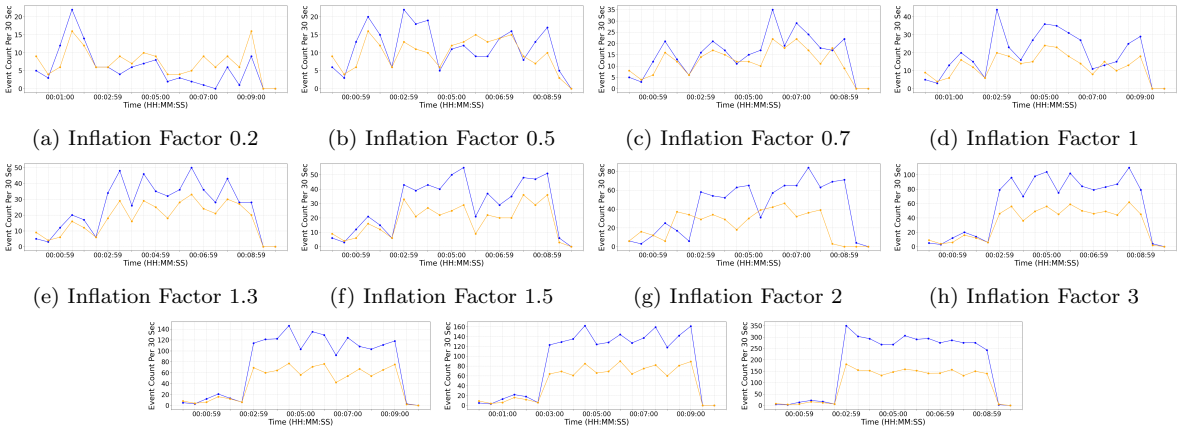


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 20: $q_2 = AND(A, B, C)$ with $\|V_{top}\| = 5$

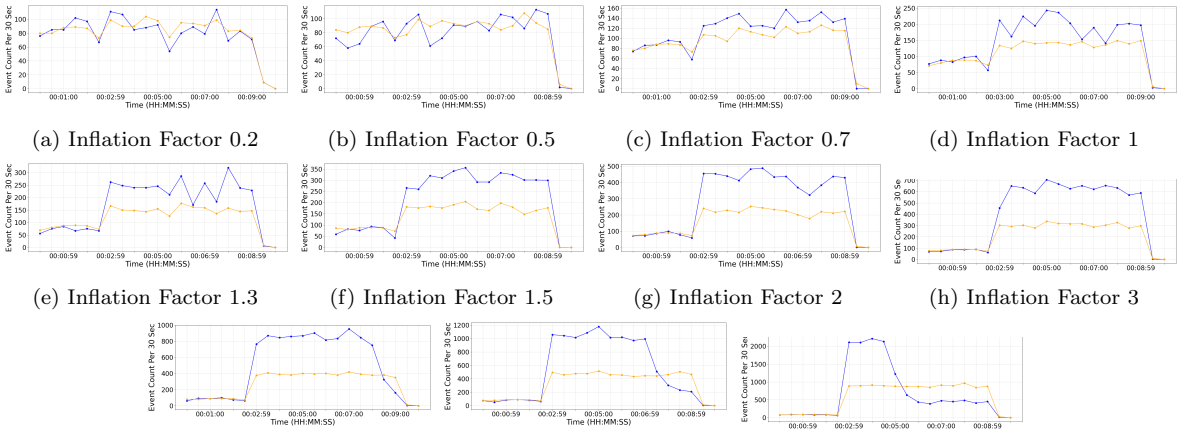


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 21: $q_2 = AND(A, B, C)$ with $\|V_{top}\| = 6$

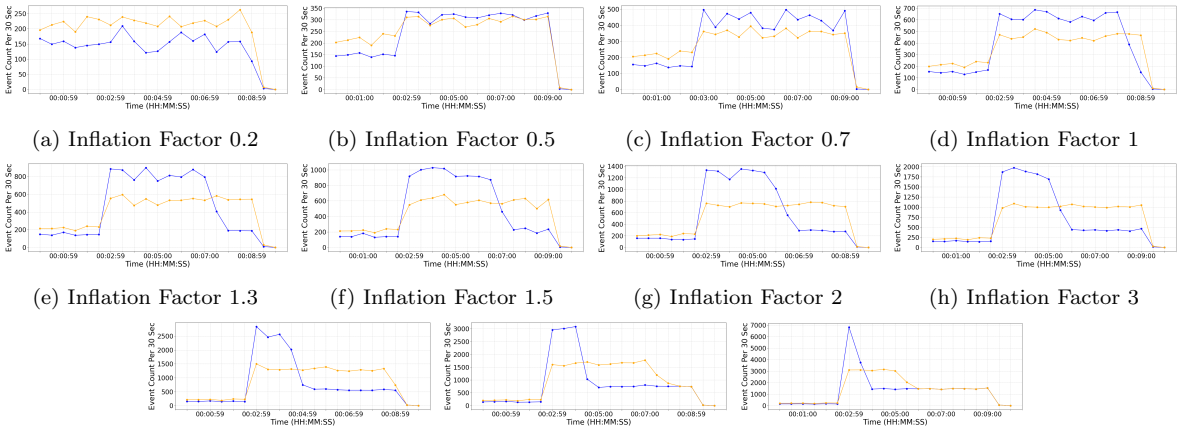


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 22: $q_2 = AND(A, B, C)$ with $\|V_{top}\| = 7$

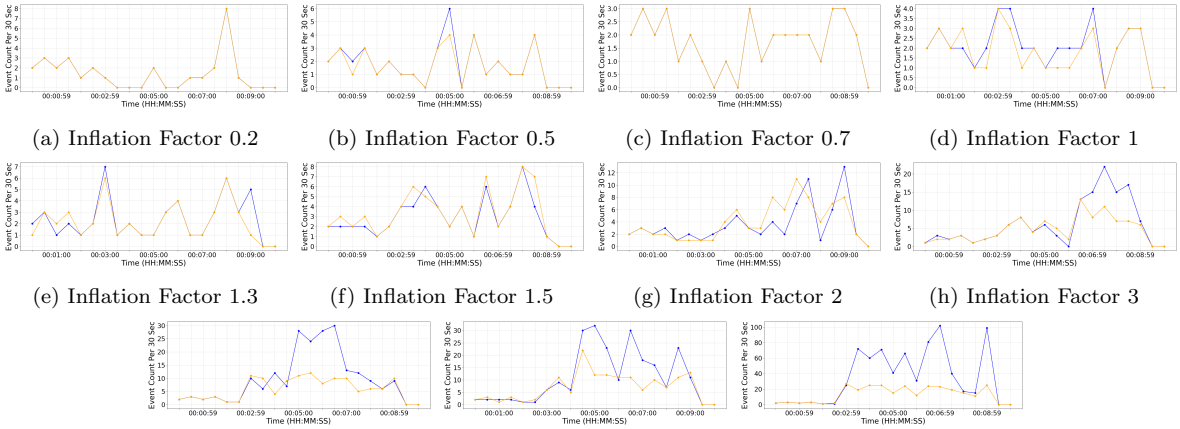


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 23: $q_2 = AND(A, B, C)$ with $\|V_{top}\| = 8$ and $AND(A, B) \in \lambda(q_2)$

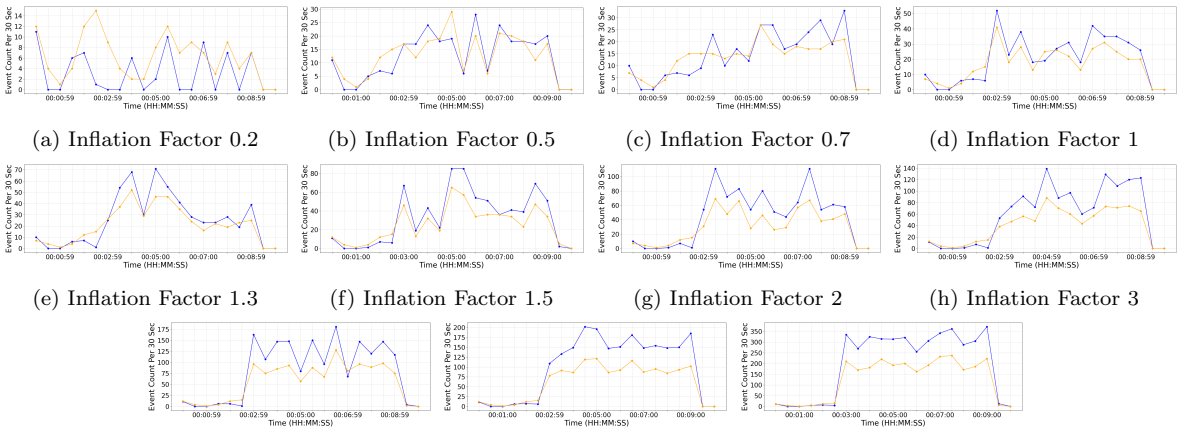


(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Figure 24: $q_2 = AND(A, B, C)$ with $\|V_{top}\| = 9$



(a) Inflation Factor 0.2 (b) Inflation Factor 0.5 (c) Inflation Factor 0.7 (d) Inflation Factor 1

(e) Inflation Factor 1.3 (f) Inflation Factor 1.5 (g) Inflation Factor 2 (h) Inflation Factor 3

(i) Inflation Factor 4 (j) Inflation Factor 5 (k) Inflation Factor 10

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den September 16, 2024 .....