Predicate-Based Push-Pull Communication for Distributed CEP

Steven Purtzel
purtzesc@hu-berlin.de
Humboldt-Universität zu Berlin
Berlin, Germany

Samira Akili
akilsami@hu-berlin.de
Humboldt-Universität zu Berlin
Berlin, Germany

Matthias Weidlich
matthias.weidlich@hu-berlin.de
Humboldt-Universität zu Berlin
Berlin, Germany

ABSTRACT

Complex event processing (CEP) enables the detection of situations of interest by evaluating queries over event streams. When applied in a networked application, events generated by distributed nodes are sent over the network to evaluate CEP queries. To reduce the transmission of events, push-based communication that sends each event immediately upon generation may be complemented with a pull-based model that buffers events locally until they are requested for query evaluation. Existing approaches that leverage push-pull communication to reduce transmission costs for distributed CEP, however, exploit solely temporal constraints imposed by a query. As such, they are not applicable in scenarios, in which relevant events may occur in each time window defined by a query.

In this paper, we propose *predicate-based push-pull* (PrePP) plans for CEP queries to overcome the above limitations. Our idea is to construct pull requests that enable fine-granular filtering at event sources based on query predicates, thereby reducing event transmission. Since the construction of optimal PrePP plans is NP-hard, we introduce a set of algorithms to speed up the plan construction by up to five orders of magnitude compared to a brute-force approach, while producing near-optimal results. In extensive experiments, we demonstrate that PrePP plans reduce event transmission by up to three orders of magnitude over baseline techniques.

CCS CONCEPTS

• **Information systems** → **Data streams**.

KEYWORDS

Complex event processing, pattern detection, distributed streams

ACM Reference Format:
Steven Purtzel, Samira Akili, and Matthias Weidlich. 2022. Predicate-Based Push-Pull Communication for Distributed CEP. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524860.3539640

1 INTRODUCTION

Complex event processing (CEP) comprises methods that enable the detection of situations of interest by continuously evaluating queries over streams of events [11]. To this end, CEP queries specify patterns of events through operators, time windows, and predicates that constrain the attribute values of events within a pattern.

In networked systems, such as IoT infrastructures or sensor networks, event streams are generated by distributed nodes. The evaluation of queries then requires transmission of events over the network. Traditional models are centralized, so that all events produced in the network must eventually be sent to a single node [2, 10]. Yet, this is rarely reasonable since, typically, only a small fraction of the events is ultimately required to generate query matches. Moreover, as query evaluation scales exponentially in the number of events to process [25], a centralized model has limited scalability.

Distributed CEP [8, 10], in turn, leverages the nodes in a network for query evaluation, assembling the matches of a query from those of sub-queries placed at the nodes [18]. Then, the communication model offers an angle for optimization: Instead of pushing all events required for query evaluation over the network immediately, pull-based communication may be exploited [2, 10, 23]. Events may be buffered locally at a node until their transmission is triggered explicitly by a request received from another node in the network. While pull-based communication implies a certain overhead in terms of storage requirements and detection latency, it has the potential to drastically reduce the event transmission.

To realize distributed CEP that relies on push-based and pull-based communication, the following questions need to be answered:

(Q1) How to model pull requests, i.e., how to select the events that shall be transmitted in response?

(Q2) When shall query evaluation adopt push-based communication and when shall events be pulled?

Existing techniques answer the above questions based on temporal constraints and the rates with which events are generated [2, 10]: A pull request fetches events of a particular type that fall into a time window of the query. As such, the decision on pull-based communication depends on the relation of the window size of a query and the generation rates of the events. Hence, these techniques are not applicable in scenarios with small time windows or high event rates. Here, pull-based communication may degenerate and fetch *all* events and, despite the induced overhead, does not reduce event transmission compared to a push-based model.

In this paper, we argue for a more expressive model for push-pull-based communication in distributed CEP to enable optimization in a wider range of application scenarios. Specifically, we exploit the query semantics in terms of the predicates that constrain the attribute values of events to achieve fine-granular selection of the events to send in response to a pull request. To realize this idea, we make the following contributions:

- We propose PrePP plans for the distributed evaluation of CEP queries, which include pull requests based on query predicates.
- We introduce a cost model for PrePP plans and show NP-hardness of the problem of computing an optimal plan.
- We present a sampling algorithm and caching strategies for the efficient construction of near-optimal PrePP plans.

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06.
https://doi.org/10.1145/3524860.3539640

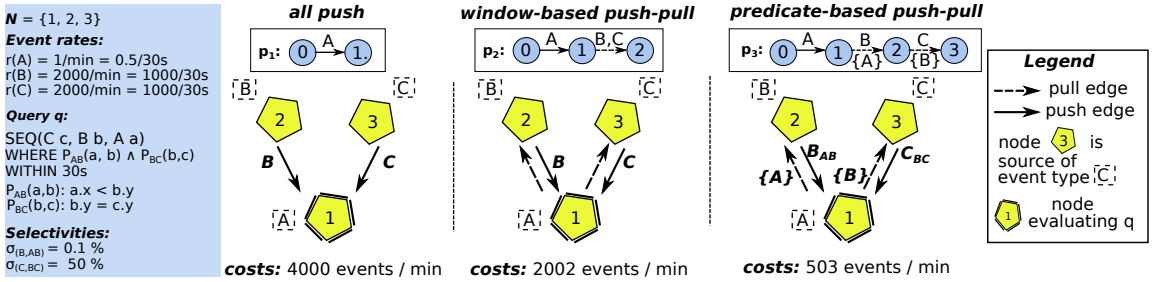

Figure 1: Event network with nodes 1-3, producing events of types A, B, and C, and three evaluation plans p_1-p_3 for query q.

We evaluated our approach with synthetic and real-world data. Our results highlight that PrePP plans reduce event transmission up to three orders of magnitude compared to baseline techniques.

Below, we motivate predicate-based push-pull communication (§2) and present a formal model for CEP (§3). We then define the problem of efficient query evaluation in networks (§4), introduce PrePP plans to address it (§5), and elaborate on the efficient construction of these plans (§6). Finally, we present our experimental evaluation (§7), review related work (§8), and conclude (§9).

2 MOTIVATING EXAMPLE

We consider networks of event producing nodes. One of the nodes is responsible for evaluating a query over the events generated by the network, such as a control station in a smart grid that monitors the energy consumption of smart households [14]. To motivate our approach, we present an abstract view of such a network in Fig. 1. Node 1 generates events of type A that occur at a low rate, i.e., once per minute. In contrast, events of types B and C are frequent, i.e., 2000 occurrences per minute, and generated at nodes 2 and 3, respectively. The given query q describes a pattern as a sequence of a C event, a B event, and an A event that occur within 30 seconds. Moreover, for events to form a match, the predicates P_{AB}, defined over A and B events, and P_{BC}, defined over B and C events, need to hold. Node 1 shall be responsible for generating matches of q.

For traditional push-based communication, as illustrated in plan p_1, all events need to be sent over the network and processed at node 1. This results in transmission costs of 4000 events per minute.

Window-based push-pull communication [2, 10], as in plan p_2, exploits the difference in the event rates to reduce event transmission. Having the lowest rate, event type A is pushed. Node 1 evaluates the query q and is the only source of A events so that no A events are sent over the network. For each A event, a pull request is sent to nodes 2 and 3 to fetch events of types B and C. In response, B and C events are sent to node 1 if they can be part of a match of q based on the constraints induced by the time window, i.e., if they occurred up to 30 seconds before the time indicated by the pull request. On average, 1000 events of types B and C satisfy the time window constraint per minute. Hence, with two pull requests, we arrive at a transmission cost of 2002 events per minute, a reduction of 50% compared to the push-based plan.

Yet, relying solely on temporal constraints for pulling events ignores the optimization potential induced by the query's predicates. They can be used to filter events before sending them in response to pull requests. This is exemplified in the predicate-based push-pull plan p_3. Again, A events are pushed. For each A event, B events are then pulled from node 2. However, the pull request no longer contains solely the timestamp of the A event, but also its attribute values required to evaluate predicate P_{AB}. In response, only those B events are sent that satisfy both, the temporal constraint and the predicate P_{AB}. With the ratio of B events that satisfy P_{AB}, i.e., the selectivity $\sigma_{(B,AB)}$, assuming a value of 0.1%, from 1000 B events in the time window, only one satisfies the predicate P_{AB}. Hence, a single event is sent in response to the pull request. Upon the arrival of a B event at node 2, C events are pulled from node 3. Here, the predicate P_{BC} is used to reduce the events in the pull response. With the selectivity of P_{BC} being 50%, from 1000 C events in the time window, 500 events are sent as a response. Hence, the transmission cost of plan p_3 is 503 events per minute; a further reduction by 75% compared to the window-based push-pull plan.

3 BACKGROUND

We define a network model and query language, as follows.

3.1 Network Model

Event Types. Let $\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_n\}$ be the universe of event types. An instantiation of an event type is called *event*, which represents an instantaneous change of state. An event type defines a schema for a set of events of similar semantics. For an event e, we write $e.t$ and $e.type$ to denote its occurrence time and event type, respectively.

Event Network. We consider an event network $\Gamma = (N, f, r)$, in which the nodes N serve as event sources and produce subsets of the event types in $\mathcal{E}$. A function $f : \mathcal{E} \to 2^N$ defines for an event type ϵ, the nodes that are capable of generating ϵ events, i.e., the sources of ϵ. A function $r : \mathcal{E} \to \mathbb{R}$ assigns to an event type the (local) rate with which the events are generated per source. The global rate $R : \mathcal{E} \to \mathbb{R}$ assigns to an event type the total number of events in the network per time unit, given as $R(\epsilon) \mapsto |f(\epsilon)| \cdot r(\epsilon)$.

Example 1. *In Fig. 1, the event network comprises nodes $N = \{1, 2, 3\}$ with $f(A) = \{1\}$, $f(B) = \{2\}$, $f(C) = \{3\}$ and the rates being $r(A) = 1$ and $r(B) = r(C) = 2000$. As each event type is generated by one source, local and global rates are equivalent.*

We consider networks with a clique structure, i.e. events can directly be exchanged between each pair of nodes.

Global and local trace. A local trace denotes the timely ordered infinite sequence of events produced by any node in an event network. Interleaving local traces of all nodes in the network, conceptually, yields a global trace that is never materialized. We assume that timestamps are sufficiently fine-grained, such that the global trace is totally ordered.

3.2 Query Language

We employ a common language model [4, 11]: A query consists of operators, a set of predicates, and a time window. Operators specify the types and order of events in query matches, which are further constrained by the query predicates and the time window. We focus on queries, for which each predicate is defined over at most two event types and for which the predicates are connected by logical conjunction (see Fig. 1). Predicates defined over single event types are directly incorporated in the characterization of event types.

Syntax. A query $q = (O, \beta)$ is defined as an ordered tree of operators, annotated with predicates and a time window. The operators O are the vertices of the query tree with the leafs being primitive operators $O_p \subseteq O$ that refer to event types. We write $o.sem$ to denote the semantic type of an operator. The semantic type of a primitive operator $o_p \in O_p$ is given by $o_p.sem \in \mathcal{E}$. A composite operator $o_c \in O_c = O \setminus O_p$ has a semantic type $o_c.sem \in \{SEQ, AND, OR\}$. The structure of query q is given by the function $\beta : O_c \twoheadrightarrow O^k$, with $k \in \mathbb{N}, k > 1$, which assigns a sequence of child operators to a composite operator. The set of event types over which q is defined is given as $\mathcal{E}(q) = \bigcup_{o_p \in O_p} o_p.sem$. We further assume that each event type is referenced by at most one primitive operator.

Semantics. While each source in the event network Γ generates a local trace of events, the semantics of a query is defined over the global trace. As such, a match of a query is given by a sequence of events of the global trace generated in Γ. The set of matches of a query $q = (O, \beta)$ is inductively defined over the operator tree:

- A primitive operator $o_p \in O_p \subseteq O$ creates a match for each occurrence of an event of type $o_p.sem$.
- An AND operator constructs a match for any interleaving of matches of all its child operators.
- A SEQ operator constructs a match for the concatenation of matches of its child operators in the specified order.
- An OR operator constructs a match for each match of one of its child operators.

However, a match is only constructed for events that satisfy the predicates and time window specified by the query. We do not impose any restrictions on the number of times an event can participate in matches, which corresponds to the most challenging evaluation scenario, also known as skip-till-any-match [1].

Rates. Given an event network $\Gamma = (N, f, r)$ the evaluation of a query $q = (O, \beta)$ can be captured by the rate with which matches of q are generated. For a primitive operators $o_p \in O_p$ with $o.sem = \epsilon$, this rate is given by $R(\epsilon)$.

We define the selectivity $\sigma_{\epsilon_i, \epsilon_j}$ as the aggregated selectivity of predicates of q defined over the event types $\epsilon_i, \epsilon_j \in \mathcal{E}(q)$. For a composite operator $o_c \subseteq O_c$, let $O_p^c \in O_p$ denote the subset of primitive operators that are leafs of the sub-tree in the query tree of q having o_c as root. The selectivity $\sigma(o_c)$ of the composite operator o_c is given by $\prod_{x \in \{(\epsilon_i, \epsilon_j) | \epsilon_i, \epsilon_j \in O_p^c \wedge \epsilon_i \neq \epsilon_j\}} \sigma_x$. For a composite operator $o_c \in O_c$ with selectivity $\sigma(o_c)$, the rate is inductively defined:

$$R(o_c) \mapsto \begin{cases} \sigma(o_c) \cdot |\beta(o_c)| \cdot \prod_{o' \in \beta(o_c)} R(o') & \text{if } o_c.sem = AND, \\ \sigma(o_c) \cdot \prod_{o' \in \beta(o_c)} R(o') & \text{if } o_c.sem = SEQ, \\ \sigma(o_c) \cdot \sum_{o' \in \beta(o_c)} R(o') & \text{if } o_c.sem = OR. \end{cases}$$

Based thereon, the rate $R(q)$ and the selectivity $\sigma(q)$ of a query q are defined as the rate and selectivity of its root operator, respectively.

Table 1: Overview of notations for networks and queries.

Notation	Explanation
ϵ	Event type
$\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_n\}$	Universe of event types
$\Gamma = (N, f, r)$	Event network: nodes N, sources of event types $f : \mathcal{E} \to 2^N$, rates $r : \mathcal{E} \to \mathbb{R}$
$r(\epsilon)$, $R(\epsilon)$	Local and global rate of an event type ϵ
$q = p = (O, \beta)$	Query q (or projection p) with composite and primitive operators $O = O_c \cup O_p$ and their tree structure $\beta : O_c \twoheadrightarrow O^k$
$\mathcal{E}(q)$, $\mathcal{E}(p)$	Event types of a query q and a projection p
Π_q	All possible projections of query q
$\pi(q, X)$	Projection of q restricted to the event types in $X \subseteq \mathcal{E}(q)$
$\sigma_{\epsilon_i, \epsilon_j}$	Selectivity of predicates defined over event types ϵ_i, ϵ_j
$\sigma(o)$, $\sigma(q)$, $\sigma(p)$	Selectivity of an operator, a query, or a projection
$\sigma(\epsilon, \pi(q, \mathcal{E}(q)'))$	Selectivity of the projection $\pi(q, \mathcal{E}(q)'))$ for events of type ϵ

Query Projection. Based on [3], we use the notion of a projection $p = (O, \beta)$ of a query $q = (O, \beta)$, which is itself a query that is restricted to a subset of the event types referenced in q, i.e., $\mathcal{E}(p) \subseteq \mathcal{E}(q)$. Moreover, the projection p of q inherits the predicates of q that are defined over $\mathcal{E}(p)$ as well as q's time window.

The set of all possible projections of a query q is Π_q. The function $\pi : q \times 2^{\mathcal{E}(q)} \to \Pi_q$ returns for a query q and a subset $\mathcal{E}(q)' \subseteq \mathcal{E}(q)$ of its event types, the projection $\pi(q, \mathcal{E}(q)')$ of q that is restricted to $\mathcal{E}(q)'$. An instantiation of function π can be found in [3].

Example 2. *For our running example in Fig. 1, $\pi(q, \{A, C\})$ yields the projection $SEQ(C, A)$, which has no predicates and the same time window as query q, i.e.,* 30 *seconds.*

We further consider the selectivity of a projection for an event type. Let q be a query defined over the event types $\mathcal{E}(q)$. Then, by $\sigma(\epsilon, \pi(q, \mathcal{E}(q)'))$, we denote the selectivity of the projection $\pi(q, \mathcal{E}(q)')$ for the event type $\epsilon \in \mathcal{E}$.

Table 1 provides an overview of our notions and notations.

4 PROBLEM STATEMENT

Next, we formalize the problem of efficient query evaluation in event networks. Let q be a CEP query to be evaluated in an event network $\Gamma = (N, f, r)$. In this context, an *evaluation plan* is a function ξ that takes a query and network as input and returns the set of matches of q at one of the nodes of the network Γ. To this end, an evaluation plan evaluates (sub-)queries at the nodes of the network and exchanges their matches to eventually assemble the matches of query q. For instance, a simple centralized, push-based evaluation plan would choose a distinguished node to evaluate the query q, while all other nodes that generate events of the types referenced in q send the events to this node.

The quality of an evaluation plan is determined by a cost $c(\xi) \in \mathbb{R}$, which describes the number of messages per time unit that need to be exchanged for producing all matches of q. This cost is based on the rates with which events are produced in the network.

Example 3. *Consider plan p_1 in Fig. 1 for the evaluation of $q = SEQ(C, B, A)$ at node* 1. *The cost of the plan is $r(B) + r(C) = 4000$.*

We summarize the problem addressed in this work, as follows.

Problem 1 (Efficient Query Evaluation in Event Networks). *Let q be a query to be evaluated in the event network $\Gamma = (N, f, r)$. The problem of* Efficient Query Evaluation in Event Networks *is to construct an evaluation plan ξ that minimizes $c(\xi)$.*

5 PREDICATE-BASED PUSH-PULL PLANS

To address the above problem of efficient query evaluation in event networks, we propose predicate-based push-pull (PrePP) plans. Below, we first present a formal model for PrePP plans (§5.1), elaborate on the execution of such a plan (§5.2), and characterize its correctness (§5.3). Then, we turn to the efficiency of PrePP plans. Based on a cost model (§5.4), we elaborate on the problem of constructing an optimal PrePP plan for query evaluation (§5.5).

5.1 PrePP Model

Our model leverages the predicates of a query defining the events to be considered in a match. That is, pull requests are enriched with some payload of events in order to enable predicate-based filtering at event sources. To realize this idea, we introduce the notion of a *pull set*, which captures the events that shall be sent in pull requests.

Example 4. *In the second step of plan p_3 in Fig. 1, only the B events that satisfy predicate P_{AB} are pulled. To enable the evaluation of this predicate at the source of B, i.e., node 2, the pull set $\{A\}$ is used, i.e., A events that are required to check P_{AB} are sent in pull requests.*

Having introduced the intuition of pull sets, we are ready to define the atomic building block of our model. To evaluate a query, nodes in the network need to acquire all relevant events that are not produced by the node itself, by either pulling them proactively or by having them pushed from their sources. We generalize the communication required for query evaluation with the notion of an *acquisition step*. It defines which types of events need to be acquired by a node for query evaluation, and potentially includes a pull set that can be used for filtering, if events are pulled instead of pushed.

Definition 1 (Acquisition Step). *Given a query q, an acquisition step $s = (\Psi_s, \rho_s) \in \Sigma = 2^{\mathcal{E}(q)} \times 2^{\mathcal{E}(q)}$ is a tuple with Ψ_s denoting the pull set used to pull events of the event types in ρ_s.*

An empty pull set of a step $s = (\Psi_s, \rho_s)$, i.e., $\Psi_s = \emptyset$, denotes that events of the types in ρ_s are pushed, i.e., sent upon their generation.

A *Predicate-based Push-Pull* (PrePP) plan for a query q evaluated in the network $\Gamma = (N, f, r)$ specifies the node $n \in N$ responsible for generating matches of q, as well as the order, in which events in $\mathcal{E}(q)$ are acquired. As such, a PrePP plan is defined as follows:

Definition 2 (PrePP Plan). *Given a query q and an event network $\Gamma = (N, f, r)$, a PrePP plan $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ is a tuple with $\hat{p}_{node} \in N$ being the node generating matches of q, and $\hat{p}_{steps} = \langle s_1, s_2, \ldots, s_n \rangle \in \Sigma^*$ being a sequence of acquisition steps.*

Example 5. *The PrePP plan $p_3 = (\hat{p}_{node}, \hat{p}_{steps})$ in Fig. 1 is given by $\hat{p}_{node} = 1$ and $\hat{p}_{steps} = \langle (\emptyset, \{A\}), (\{A\}, \{B\}), (\{B\}, \{C\}) \rangle$. The first step $(\emptyset, \{A\})$ indicates that A events are pushed. In the second step, $(\{A\}, \{B\})$, events of type A are used to pull events of type B.*

Below, we need some auxiliary notion for the types of events that are available already when conducting an acquisition step. Let $\hat{p}_{steps} = \langle s_1, s_2, \ldots, s_n \rangle$ be the acquisition steps of a PrePP plan for a query q. Then, $\mathcal{A} : \Sigma \rightarrow 2^{\mathcal{E}(q)}$ assigns all event types acquired in earlier steps to an acquisition step s_j, $1 \leq j \leq n$, of $\hat{p}_{steps}$:

$$\mathcal{A}(s_j) \mapsto \bigcup_{\substack{1 \leq k < j, \text{ with} \\ \hat{p}_{steps} = \langle s_1 = (\Psi_1, \rho_1), s_2 = (\Psi_2, \rho_2), \ldots, s_n = (\Psi_n, \rho_n) \rangle}} \rho_k .$$

Table 2: Overview of notations for PrePP model.

Notation	Explanation
Ψ_s	Pull set
ρ_s	Events to pull
$s = (\Psi_s, \rho_s)$	Acquisition step with pull set Ψ_s and event types to pull ρ_s
$\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$	PrePP plan: node $\hat{p}_{node}$ executes acquisition steps $\hat{p}_{steps}$
$\mathcal{A}(s)$	Set of event types already acquired in step s

Example 6. *For the acquisition step $s_3 = (\{B\}, \{C\})$ of the plan p_3 in Fig. 1, the set of available event types is $\mathcal{A}(s_3) = \{A, B\}$.*

5.2 PrePP Plan Execution

We now explain the evaluation of a query q with a PrePP plan $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ in the network $\Gamma = (N, f, r)$.

Input Streams. Each node in N processes two input streams: the *input event stream*, which comprises events required to generate matches of the query, and the *input request stream*, which contains pull requests. As $\hat{p}_{node}$ is responsible for generating matches of q, its input event stream contains locally generated events as well as the events sent by other nodes in the network in response to pull requests. Note that events in the input event stream of $\hat{p}_{node}$ may need to be ordered by their timestamps. The input event stream of all other nodes in $N \setminus \{\hat{p}_{node}\}$ corresponds to their local trace.

Partial Match-based Query Evaluation. To generate matches of query q, node $\hat{p}_{node}$ employs an evaluation algorithm that processes the input event stream to build partial matches that may eventually become complete matches of q. While we abstract from the actual evaluation algorithm used by $\hat{p}_{node}$, it must be capable of processing events out of order.

The sources of events that are pulled during query evaluation handle pull requests as follows: Let ϵ be an event type that is pulled using the pull set Ψ. To enable predicate-based filtering, matches of the projection $p = \pi(q, \Psi \cup \{\epsilon\})$ are generated based on the events received in pull requests and the locally generated ϵ events. We denote a partial match of p comprising only events of the types in Ψ as an *active pull request*.

Buffers. For each event type ϵ that is pulled, the respective sources maintain a buffer comprising locally generated events of type ϵ as well as the active pull requests for ϵ. Moreover, $\hat{p}_{node}$ maintains a buffer containing the set of partial matches generated during the evaluation of query q.

Based thereon, we characterize the query evaluation at any network node using two functions, *step* and *answer*:

$$step : e, B \mapsto B', M, R, E$$
$$answer : r, B \mapsto B', E$$

Step. The function *step* takes as input an event e of the input event stream of node n as well as the node's current buffer state B. It returns a new buffer state B', a set of complete matches M of the query, a set of pull requests R, and events to be sent E.

For event e, let there by an acquisition step $s_i = (\Psi_i, \rho_i)$ in the PrePP plan, such that $e.type \in \rho_i$. If the step function is evaluated at a node $n \in N \setminus \{\hat{p}_{node}\}$ and $\Psi_i = \emptyset$, e is pushed to $\hat{p}_{node}$. Otherwise, i.e., $\Psi_i \neq \emptyset$, e is added to the buffer of node n. If the step function is evaluated at $\hat{p}_{node}$, e is processed by the query evaluation algorithm, which may create new partial matches. If a new partial match consisting of events of all types in $\mathcal{A}(s_i)$ is

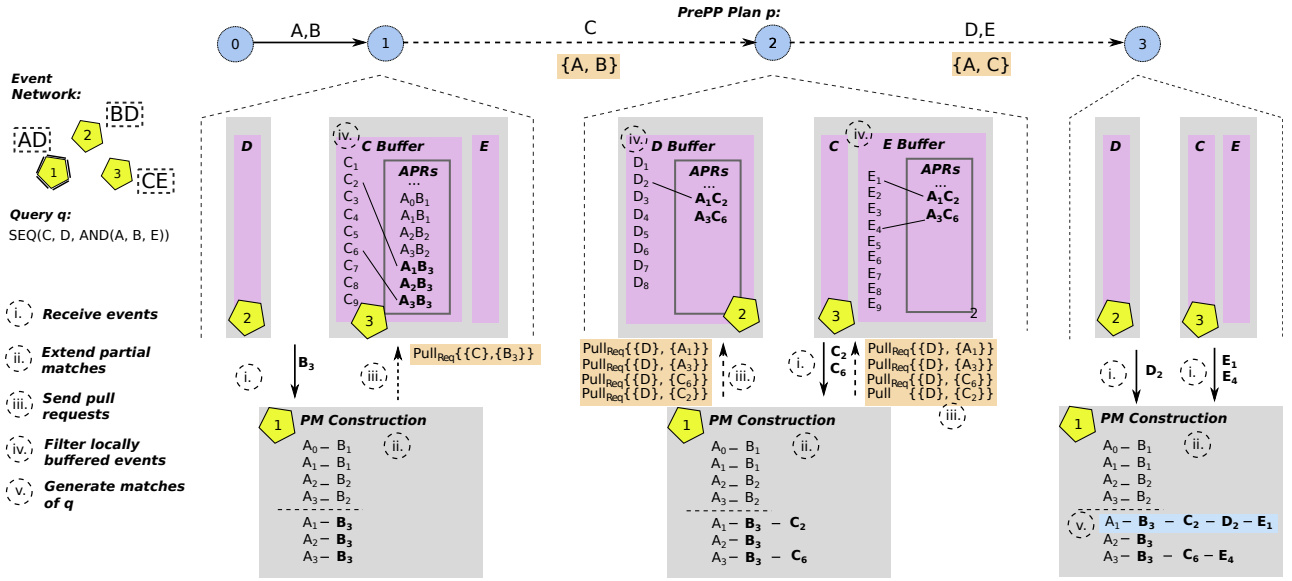

Figure 2: Execution of the PrePP Plan $\hat{p}$ (shown on top) for the query q evaluated in the event network Γ. The progress of evaluating q at node 1 as well as the buffer states of node $1-3$ after each of the acquisition steps of $\hat{p}$. Arrows between the three network nodes denote the exchange of pushed events, pull requests and events sent as pull responses.

created while processing e, a pull request is sent for each event in the partial match of a type that is referenced in the pull set Ψ_{i+1} of the (next) acquisition step $s_{i+1} = (\Psi_{i+1}, \rho_{i+1})$. This way, all events of types ρ_{i+1} are pulled from their sources.

Answer. The function *answer* takes as input a pull request r of the input request stream and the node's current buffer state B. It returns a new buffer state B' and set of events E to be sent in pull responses. Let r be an event received in a pull request based on the pull set Ψ used to pull events of type ϵ. The event r is processed by the evaluation algorithm used to evaluate the projection $p = \pi(q, \Psi \cup \{\epsilon\})$. For each new match of the projection p generated based on event r, the contained ϵ event is sent as a pull response and the partial matches of p comprising only events of the types referenced in the pull set Ψ are buffered as active pull requests. To ensure correct evaluation, an active pull request is maintained as long as an ϵ event can be generated that, together with events of this active request, leads to a new match of the projection $\pi(q, \Psi \cup \{\epsilon\})$.

Example 7. *We illustrate the execution of a PrePP plan with the example shown in Fig. 2. Given the illustrated network with three nodes, for a query $q = SEQ(C, D, AND(A, B, E))$, a PrePP plan $\hat{p}$ with $\hat{p}_{node} = 1$ and $\hat{p}_{steps} = \langle(\emptyset, \{A, B\}), (\{A, B\}, \{C\}), (\{A, C\}, \{D, E\})\rangle$ is executed. Here, node 1 generates the matches of query q and, therefore, maintains a buffer of partial matches of q. The other nodes, in turn, maintain a buffer for the events that are generated locally and that are pulled according to the evaluation plan. That is, node 2 buffers events of type D and node 3 buffers those of types $\{C, E\}$.*

In the first step of $\hat{p}$, A and B events are pushed to node 1. As events of type A are generated only by node 1, events of type A do not have to be sent over the network. The figure illustrates the state after an event B_3 has been received by node 1, which led to the creation of new partial matches $(A1, B3)$, $(A2, B3)$, and $(A3, B3)$. As a consequence, pull requests are triggered. However, in the example, the A events of the new partial matches have already been sent (i.e., events A_0-A_3 are contained in the active pull requests (APRs) at node 3). Hence, only B_3 is sent to node 3 in a request to pull C events.

Node 3 receives the request and constructs matches of the projection $\pi(q, \{A, B, C\}) = SEQ(C, AND(A, B))$. The C events contained in matches of $SEQ(C, AND(A, B))$, in this case C_2 and C_6, are sent as pull responses. Moreover, the partial matches of $SEQ(C, AND(A, B))$ comprising only events of the pull request, i.e., A and B events, denote active pull requests and are stored in the buffer.

Node 2 receives events C_2 and C_6, which leads to the creation of two new partial matches, comprising the events of available types given as $\mathcal{A}((\{A, B\}, \{C\})) = \{A, B, C\}$. This, in turn, triggers the pull requests of the next acquisition step.

In the next (and last) acquisition step, the pull set is given by $\{A, C\}$. Therefore, a pull request is sent for each A and C event in the newly created partial matches, i.e., C_2, C_6, A_3, A_1, to the sources of D and E events. Based on the events received in the pull requests, node 2 and node 3 generate matches of the projections $SEQ(C, D, A)$ and $SEC(C, AND(A, E))$, respectively. Then, D events and E events in the resulting matches are sent in pull responses.

Receiving the events D_2, E_1, E_4 at node 1 completes the execution of the last step of $\hat{p}$. It generates the match $(A_1, B_3, C_2, D_2, E_1)$ of q.

5.3 Correctness of PrePP Plans

A PrePP plan must guarantee correct query evaluation. That is, for a PrePP plan $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ for the evaluation of q in a network $\Gamma = (N, f, r)$, the set of matches generated at $\hat{p}_{node}$ must be equivalent to the set of matches of q that can be generated over the global trace of the network. Assuming that the employed query evaluation algorithm operates correctly, for the PrePP plan $\hat{p}$ to be correct, it must hold that pull sets are based only on available event types and that all event types are eventually acquired. We capture these requirements as follows:

Definition 3 (Correct PrePP Plan). *Let $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ be a PrePP plan with $\hat{p}_{steps} = \langle s_1, s_2, \ldots, s_n \rangle$ for the evaluation of query q. Plan $\hat{p}$ is correct if*

(i) the pull set of each acquisition step is a subset of the available event types, $\forall\ 1 \leq k \leq n, s_k = (\Psi_k, \rho_k) : \Psi_k \subseteq \mathcal{A}(s_k)$; and

(ii) all events necessary to generate the query matches are acquired, $\mathcal{E}(q) = \bigcup_{1 \leq k \leq n,\ s_k = (\Psi_k, \rho_k)} \rho_k$.

5.4 Cost Model

We quantify the quality of a PrePP plan for a query q in terms of the induced transmission costs. This cost corresponds to the sum of the rates with which the events required to generate matches of q are sent over the network. For our cost model, we consider the total rate $R(\epsilon)$ of an event type ϵ per time window specified by q. As such, we abstract from the size of messages used to send events which may vary based on the respective event type or the type of message, i.e., pull requests or pull responses. Moreover, we assume that direct communication between the nodes is possible such that the costs of exchanging events is the same for each pair of nodes.

For a PrePP plan $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$, we define a function $c : \Sigma \rightarrow \mathbb{R}$ that assigns transmission costs to an acquisition step $s = (\Psi_s, \rho_s)$ of $\hat{p}_{steps}$, as follows:

$$c(s) \mapsto \sum_{\epsilon \in \rho_s} |f(\epsilon)| \cdot \sum_{\epsilon \in \Psi_s} R(\epsilon)\sigma_{(\epsilon, \mathcal{A}(s))} + \sum_{\epsilon \in \rho_s} R(\epsilon)\sigma_{(\epsilon, \{\epsilon\} \cup \Psi_s)}.$$

The first part of the sum captures the rates with which pull requests are sent. The left factor defines the number of sources producing events of the types ρ_s to acquire, as each of those receive pull requests in step s. A pull request is sent whenever an event referenced by the pull set Ψ_s is received that triggers the creation of a new partial match comprising events of all types in $\mathcal{A}(s)$. The latter is equivalent to the generation of a new match of the projection $\pi(q, \mathcal{A}(s))$. As such, for each event type $\epsilon \in \Psi_s$, the selectivity $\sigma(\epsilon, \pi(q, \mathcal{A}(s)))$ describing the portion of distinct ϵ events in matches of projection $\pi(q, \mathcal{A}(s))$ estimates the rate with which ϵ events are sent as pull requests.

The second part describes the rates with which pull responses are sent. Let $\epsilon \in \rho_s$ be an event type to be acquired with the pull set Ψ_s. To filter the ϵ events at their sources before sending them back in respective pull responses, matches of the projection $p = \pi(q, \{\epsilon\} \cup \Psi_s)$ must be generated. The distinct ϵ events in matches of p, estimated by applying the selectivity $\sigma(\epsilon, \{\epsilon\} \cup \Psi_s)$ to ϵ's rate, are sent in the pull responses.

The total cost of a PrePP plan $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ is then obtained by aggregating the costs of its acquisition steps:

$$c(\hat{p}) \mapsto \sum_{\substack{1 \leq k < n, \text{ with} \\ \hat{p}_{steps} = \langle s_1, s_2, \ldots, s_n \rangle}} c(s_k) - \sum_{\substack{\epsilon \in \mathcal{E}(q), \text{ s.t.} \\ \hat{p}_{node} \in f(\epsilon)}} r(\epsilon).$$

Here, the events generated by $\hat{p}_{node}$ required for the evaluation of query q do not have to be sent over the network and, hence, are subtracted from the costs of the acquisition steps.

5.5 Optimality of PrePP Plans

The above cost model induces a notion of optimality for a PrePP plan to evaluate a query q in an event network Γ.

Definition 4 (Optimal PrePP Plan). *Let $\hat{p} = (\hat{p}_{node}, \hat{p}_{steps})$ be a correct PrePP plan for the evaluation of a query q in an event network Γ. Then, the plan $\hat{p}$ is optimal if it has minimal cost, i.e., there exists no other correct PrePP plan $\hat{p}_i$ for q in Γ, such that $c(\hat{p}_i) < c(\hat{p})$.*

The construction of an optimal PrePP plan is computationally hard. In fact, already the construction of a single acquisition step is challenging. Given a set of available event types, the problem to decide whether further event types needed for query evaluation shall be pushed or pulled turns out to be NP-complete. Put differently, a subset of the available event types may be used as a pull set to reduce the rates with which the events of the missing types are sent over the network. Yet, deciding whether this pays off, in comparison to pushing the respective events directly, is an NP-complete problem. We formalize this problem, induced by the construction of a single acquisition step, as follows:

Problem 2 (Pull Acquisition). *Let q be a query and $\Gamma = (N, f, r)$ be an event network. Consider the construction of an acquisition step $s = (\Psi_s, \rho_s)$ in a PrePP plan for q in Γ to acquire events of types $\rho_s \subseteq \mathcal{E}$ when the set of available event types is given by $\mathcal{A}(s) \subseteq 2^{\mathcal{E}}$. Then, the problem of* Pull Acquisition *is to decide whether there is a pull set $\Psi_s \subseteq \mathcal{A}(s)$, such that*

$$\sum_{\epsilon \in \Psi_s} r(\epsilon)\sigma_{(\epsilon, \mathcal{A}(s))} + \sum_{\epsilon \in \rho_s} r(\epsilon)\sigma_{(\epsilon, \{\epsilon\} \cup \Psi_s)} < \sum_{\epsilon \in \rho_s} r(\epsilon).$$

Theorem 1. *Pull Acquisition is NP-complete.*

Proof. *NP-Hardness:* We provide a polynomial-time reduction of Vertex Cover to Pull Acquisition. Vertex Cover is an NP-complete problem [15], where an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$ are given. It must then be decided whether there exists a subset of vertices $V' \subseteq V$ of size at most $|V'| \leq k$, such that each edge $e \in E$ has an endpoint in V'.

Our construction for this reduction, without loss of generality, ensures the following properties:

(1) For any event type $\epsilon \in \Psi_s$, it holds that $R(\epsilon) = r(\epsilon)$, meaning that only one source generates ϵ events.
(2) We pull a single event type, which is denoted by $\bar{\epsilon}$, i.e., $\rho_s = \{\bar{\epsilon}\}$.
(3) The rate of any event type $\epsilon \in \mathcal{A}(s)$ is $r(\epsilon) = 1$, while the rate $r(\bar{\epsilon})$ of the event type to be acquired satisfies $r(\bar{\epsilon}) \geq |\mathcal{A}(s)| + 1$.
(4) For any event type $\epsilon \in \Psi_s$, the selectivity $\sigma_{(\epsilon, \mathcal{A}(s))}$ is one. Hence, it follows $\sum_{\epsilon \in \Psi_s} r(\epsilon)\sigma_{(\epsilon, \mathcal{A}(s))} = \sum_{\epsilon \in \Psi_s} r(\epsilon) = |\Psi_s|$.

Consider an instance of Vertex Cover given by an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$. To construct an instance of Pull Acquisition, first, for each vertex $v \in V$, we create an event type ϵ, which yields a universe of event types $\mathcal{E}$ of size $|V|$. The resulting mapping is captured by a bijection $\mu : \mathcal{E} \rightarrow V$. Moreover, we assign each event type $\epsilon \in \mathcal{E}$ a rate of $r(\epsilon) = 1$ and define all event types as being available, i.e., $\mathcal{A}(s) = \mathcal{E}$.

Now, we add an isolated vertex $\bar{v}$ to the graph. As this vertex has a degree of zero, it is never included in a vertex cover. For $\bar{v}$, we create the event type $\bar{e}$ which, as mentioned above, has a rate satisfying $r(\bar{e}) \geq |\mathcal{A}(s)| + 1$ and is the event type to be acquired, i.e., $\rho_s = \{\bar{e}\}$.

Next, we define a function $\varphi : V \rightarrow 2^E$ that returns for a vertex $v \in V$ the set of edges being incident to v. Function φ is computable in polynomial time, e.g., by building the incidence matrix for G in $O(V^3)$ time and checking the respective row.

Let $\Psi_s \subseteq \mathcal{A}(s)$ be a pull set and let $K = \bigcup_{\epsilon \in \Psi_s} \varphi(\mu(\epsilon))$ be a set of all edges in E corresponding to the choice of the pull set Ψ_s. Again, we note that K can be calculated in polynomial time.

Let $\tilde{c} : E \times 2^E \rightarrow \{0, 1\}$ be a cost function that assigns a cost of zero, if an edge $e \in E$ is contained in K; and a cost of one, otherwise.

$$\tilde{c}(e, K) \mapsto \begin{cases} 0, & \text{if } e \in K, \\ 1, & \text{otherwise.} \end{cases}$$

We use function $\tilde{c}$ to define the selectivity $\sigma_{(\bar{\epsilon}, \{\bar{\epsilon}\} \cup \Psi_s)}$, in polynomial time, for the event type to acquire, $\bar{\epsilon}$, and a pull set $\Psi_s \subseteq \mathcal{A}(s)$:

$$\sigma_{(\bar{\epsilon}, \{\bar{\epsilon}\} \cup \Psi_s)} = \begin{cases} \frac{1}{r(\bar{\epsilon})+1}, & \text{if } \sum_{e \in E} \tilde{c}(e, K) = 0 \wedge |\Psi_s| \leq k, \\ 1, & \text{otherwise.} \end{cases}$$

Again, recall that the selection of the pull set Ψ_s determines K.

The selectivity $\sigma_{(\bar{\epsilon}, \{\bar{\epsilon}\} \cup \Psi_s)}$ is one, if there is at least one edge $e \in E$, such that $\{e\} \cap K = \emptyset$, whereby either K does not represent a vertex cover in G or it holds $|\Psi_s| > k$. The selectivity is $\frac{1}{r(\bar{\epsilon})+1}$, if $K = E$ and $|\Psi_s| \leq k$. This is the case, if the edges K corresponding to the pull set Ψ_s represent a vertex cover in G of the size $|\Psi_s| \leq k$.

To explain the value $\frac{1}{r(\bar{\epsilon})+1}$, recall the inequality of Problem 2:

$$\sum_{\epsilon \in \Psi_s} r(\epsilon)\sigma_{(\epsilon, \mathcal{A}(s))} + \sum_{\epsilon \in \rho_s} r(\epsilon)\sigma_{(\epsilon, \{\epsilon\} \cup \Psi_s)} < \sum_{\epsilon \in \rho_s} r(\epsilon).$$

Our construction yields $\rho_s = \{\bar{\epsilon}\}$ and we ensured $r(\epsilon) = 1$ and $\sigma_{(\epsilon, \mathcal{A}(s))} = 1$ for all $\epsilon \in \mathcal{A}(s)$. Hence, we obtain:

$$|\Psi_s| + r(\bar{\epsilon})\frac{1}{r(\bar{\epsilon}) + 1} < r(\bar{\epsilon}).$$

We further required $r(\bar{\epsilon}) \geq |\mathcal{A}(s)| + 1$, so that the left term of the inequality reduces to $|\Psi_s| + \frac{\mathcal{A}(s)+1}{\mathcal{A}(s)+1+1} = |\Psi_s| + 0.\overline{9}$. From $r(\bar{\epsilon}) \geq |\mathcal{A}(s)| + 1$, it also follows that $|\Psi_s| + 0.\overline{9} \leq |\mathcal{A}(s)| + 0.\overline{9} < |\mathcal{A}(s)| + 1$. Therefore, the inequality of Problem 2 holds true.

Note that calculating $\sigma_{(\bar{\epsilon}, \{\bar{\epsilon}\} \cup \Psi_s)}$ is possible in polynomial time, as for each edge $e \in E$, we check if it is contained in K.

It follows that there exists a subset $\Psi_s \subseteq \mathcal{A}(s)$ satisfying the above conditions, if and only if:

$\iff$ $\sum_{\epsilon \in \Psi_s} r(\epsilon) + \frac{r(\bar{\epsilon})}{r(\bar{\epsilon})+1} < r(\bar{\epsilon})$ and $\sum_{\epsilon \in \Psi_s} r(\epsilon) \leq k$.
Note: From $\sum_{\epsilon \in \Psi_s} r(\epsilon) \leq k$, it follows that $|\Psi_s| \leq k$, as $\sum_{\epsilon \in \Psi_s} r(\epsilon) = |\Psi_s|$.

$\iff$ $\sigma_{(\bar{\epsilon}, \{\bar{\epsilon}\} \cup \Psi_s)} = \frac{1}{r(\bar{\epsilon})+1}$.
Note: From $\sum_{\epsilon \in \Psi_s} r(\epsilon) < r(\bar{\epsilon})$, it follows $\sum_{\epsilon \in \Psi_s} r(\epsilon) + \frac{r(\bar{\epsilon})}{r(\bar{\epsilon})+1} < r(\bar{\epsilon})$.

$\iff$ Each edge $e \in E$ is contained in K and $|\Psi_s| \leq k$.
Note: The selection of the pull set $\Psi_s \subseteq \mathcal{A}(s)$ determines K.

$\iff$ The subset $\Psi_s \subseteq \mathcal{A}(s)$ corresponds to a vertex cover in G of size $|\Psi_s|$.

$\iff$ There exists a vertex cover of size at most k in G.

From the above, we conclude on NP-hardness of Pull Acquisition.

NP-Membership: Membership in NP follows from the possibility to verify a potential solution in polynomial time. That is, the inequality of Problem 2 is checked directly for some given input.

We conclude that Pull Acquisition is NP-complete. □

Finally, we note that the above results enable immediate conclusions on the corresponding optimization problem. That is, finding a pull set that incurs minimal cost must be NP-hard. Consequently, also the computation of an optimal PrePP plan for a given query and event network must be NP-hard.

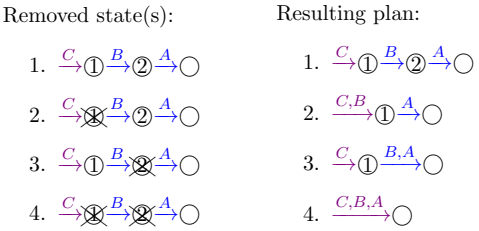

Figure 3: Partitioning Phase.

6 PREPP PLAN CONSTRUCTION

In the light of the complexity results for the construction of optimal PrePP plans, this section is devoted to the efficient computation of near-optimal plans. To this end, we first introduce a sampling algorithm (§6.1) to deal with the combinatorial nature of the plan construction, before turning to two caching strategies (§6.2).

6.1 Sampling Algorithm

In the construction of a PrePP plan, the order in which events are acquired as well as the pull sets used per step need to be determined.

PrePP Plan Ordering. The number of possible orderings to acquire events, referred to as PrePP plan orderings, corresponds to the number of all possible weak orderings for a set of n elements [2]. In [5] it was shown that $\frac{n!}{2 \cdot ln(2)^{n+1}}$ is the closest approximation of this number for $n \leq 15$. Note that the denominator $2 \cdot ln(2)^{n+1}$ is smaller than one for all $n > 0$, so that the number of PrePP plan orderings grows faster than the factorial function. As enumerating *all* possible orderings quickly becomes infeasible, we start by enumerating all single-step plan orderings, i.e., plans in which events of only one type are acquired per step. This decreases the number of possible plans to $n!$. Yet, as enumerating all possible single-step orderings also becomes infeasible for growing query lengths, we adopt sampling and randomly draw a set of s single-step orderings.

Pull Set Enumeration. To generate an optimal single-step PrePP plan for a sampled ordering, we compute the optimal pull set for each acquisition step. Given a set of available event types, $\mathcal{A}(s)$, and a set of event types to acquire, ρ_s, determining the optimal pull set $\Psi_s \subseteq \mathcal{A}(s)$ for the step $s = (\Psi_s, \rho_s)$ is NP-hard, so that a brute-force search cannot be avoided in the worst case. As such, for a single-step ordering, we enumerate all possible pull sets and compute the costs of the resulting PrePP plan, keeping for each ordering the pull sets for each step that minimize costs.

Top–k Pruning. Before transforming single-step PrePP plans to multi-step PrePP plans, in which multiple event types can be acquired within one acquisition step, we restrict ourselves to the top-k single-step plans with the lowest costs. We later provide experimental evidence that even for small values of k, the resulting PrePP plans are often close to optimal plans.

Partitioning. For each top-k single-step plan, we generate all possible multi-step plans. To this end, two adjacent acquisition steps $s_i = (\Psi_i, \rho_i)$ and $s_{i+1} = (\Psi_{i+1}, \rho_{i+1})$ in a single-step plan are merged to a step $s_i = (\Psi_i, \rho_{s_i} \cup \rho_{s_{i+1}})$. A single-step plan ordering of length n contains $n - 1$ acquisition steps, in which events are pulled. Hence, there are $2^{n-1} - 1$ respective multi-step orderings.

Example 8. *Fig. 3 illustrates the partitioning for the acquisition order of a single-step plan, which first pushes C events, before pulling*

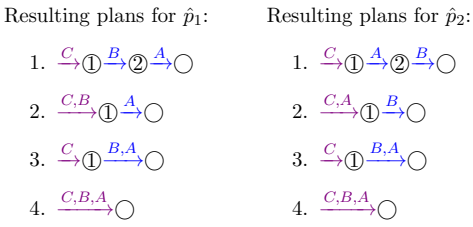

Figure 4: Caching opportunities during partitioning phase.

B events, and then pulling A events. As events are pulled in two out of three steps, there are $2^{3-1}-1=3$ possible multi-step plans.

Considering the overall complexity of our sampling algorithm, we first note that there are exponentially many multi-step plan orderings for each single-step plan ordering, while for each of them, we also need to compute the optimal pull set.

Yet, with s as the size of sample of single-step orderings and k as the pruning parameter, our sampling algorithm runs in $O(s \cdot 2^{|\mathcal{E}(q)|} + k \cdot 2^{|\mathcal{E}(q)|} \cdot 2^{|\mathcal{E}(q)|})$ and, hence, $O(k \cdot 2^{2|\mathcal{E}(q)|})$ time. As such, we avoid the aforementioned factorial growth of the runtime.

6.2 Caching Strategies

We further improve the runtime of our sampling algorithm with two caching strategies.

Pull Step Caching. For a set of PrePP plan orderings for the same query, the computation of pull sets may benefit from caching. The reasoning being that for different plans, we may assume the same state in terms of the set of available event types and the set of event types to acquire. Therefore, we compute the respective pull set only once and cache it for reuse in other plans.

Example 9. *Consider the following two plans for the same query: $\xrightarrow{A}①\xrightarrow{B}②\xrightarrow{C}③\xrightarrow{D}◯$ and $\xrightarrow{A,B}①\xrightarrow{C}②\xrightarrow{D}◯$. The first plan pushes A events, before pulling B events, C events, and D events, in that order. The second plan pushes A and B events, before pulling C and D events. In either plan, at some point, pull sets to acquire C events need to be computed with events of types $\{A, B\}$ being available. Analogously, at another point, pull sets to acquire D events are calculated based on events of types $\{A, B, C\}$ being available in either plan. As such, the pull sets shall be cached.*

Plan Caching. We employ a second caching strategy to prevent the repeated calculation of entire plans. It exploits that the top-k multi-step plan orderings generated in the partitioning phase may overlap in their structure, as similar orderings might lead to similar (low) costs. Therefore, we cache results for entire plans.

Example 10. *Consider the single-step plans $\hat{p}_1$ and $\hat{p}_2$, for which the ordering of steps is given in Fig. 4. Here, two multi-step plans are actually equivalent, so that the computed pull sets shall be cached.*

7 EXPERIMENTAL EVALUATION

We evaluated PrePP plans for distributed CEP in several experiments. Below, we first review our setup (§7.1). We then report on our results from a simulation study based on synthetic data (§7.2), before turning to a case study using two real-world datasets (§7.3).

7.1 Experimental Setup

Evaluation Plan Generation. We considered several strategies to generate an evaluation plan for a given query and event network.

PrePP. We computed the proposed PrePP evaluation plans using sampling (§6.1) and our caching strategies (§6.2). Unless pointed out otherwise, we set $s = 1024$ and $k = 10$ for the sampling algorithm.

Greedy PrePP. As a simple baseline, we implemented a greedy algorithm to compute single-step PrePP plans. Here, the event types are acquired in the order of their rates, starting with lowest rate. While the first event type is pushed, the remaining steps are computed by exploring the possible pull sets. However, to avoid the exponential blow-up of the candidate pull sets, the greedy algorithm incorporates only plans with pull set sizes of at most three.

Exact PrePP. To calculate optimal PrePP plans, we enumerated *all* possible push-pull acquisition orderings, based on which optimal pull sets have been determined. While the algorithm employs our proposed pull step caching, see §6.2, it shows a high runtime and hence, was included only in some of the experiments.

Push-Pull MuSE. To study the potential of incorporating predicate-based push-pull communication into existing algorithms for in-network processing of CEP queries, we integrated PrePP plans in MuSE graphs [3]. A MuSE graph splits a query into projections and places the projections at network nodes. In our extension, we computed a PrePP plan for each of the used projections, which is then evaluated by the respective node hosting the projection.

PPoP. As a state-of-the-art technique for distributed CEP with push-pull-based communication, we considered the approach presented by Flouris et al. [10]. It combines a placement of query operators with a model that incorporates pulling of events based on temporal constraints derived from a query time window.

Datasets. To achieve a controlled setup in the simulation experiments, we generated event networks with the following parameters:

- The *event node ratio* denotes the average percentage of event types generated by a node. We vary it from 0.1 to 1.0.
- The *event skew* models differences in the rates. We draw rates for event types from a Zipfian distribution, where a Zipfian parameter 1.1 denotes the highest differences in the rates, while, for 2.0, the rates for each event type are almost equal and low.
- The *number of nodes* captures the size of the event network, varied between 10 and 250.

As default values, we choose an event node ratio of 0.5, an event skew of 1.3, and a network of 20 nodes.

Moreover, we conducted a case study with two real-world datasets from the domains of urban transportation and cluster monitoring. We discuss the characteristics of these datasets in §7.3.

Query workload. We generated different query workloads for each set of experiments, which differ in their number of contained queries and query length. To generate a query of a given length l, i.e., based on l event types, we randomly choose a nesting depth from $[1, l-1]$ and then assign sequence and conjunction operators in alternating order to each level of nesting. Finally, we randomly disseminate the l event types among the query operators. To simulate query predicates, we uniformly drew selectivities from a predefined range and assigned them to pairs of event types. The range had a fixed maximum of 10^{-2}, while the *minimal selectivity* was used as a parameter that is varied between 10^{-6} and 10^{-2}.

Metrics. To assess the quality of the constructed evaluation plans, we measure the *transmission ratio*. It is a normalized measure defined as the transmission costs induced by the constructed plan divided by the transmission costs of a centralized push-based plan. Also, we report on the *execution time* of plan construction and measure the detection *latency* induced by the evaluation plans.

Simulation. We implemented the algorithms mentioned above in Python to conduct simulation experiments. For each network parameter, we generated 50 networks per parameter value. Moreover, we averaged the presented results over 500 runs per experiment.

Case Study Implementation. Moreover, to conduct our case study, we implemented an automata-based CEP engine, written in *C#*, which supports the evaluation of CEP queries using PrePP plans. That is, each node runs an instance of the engine, whereby the communication between nodes is handled by Ambrosia [12], a framework for resilient distributed computing.

Watermarking. To evaluate PrePP plans, events, active pull requests, and partial matches must be buffered as long as they can lead to the generation of a correct match. However, due to the distributed nature of our setting, it cannot be decided for a single event if it can still lead to a match by solely considering the time window of the query. Therefore, we realize a watermarking mechanism: While we cannot make assumptions about the time in which another node receives an event generated by a node, we assume that all events sent by one node are received in the order in which they have been sent. Exploiting this order, we keep track of the newest timestamp carried by an event for each node so that the events received from all nodes enable us to derive a global watermark. Based thereon, we can remove items that may no longer participate in a match from the respective buffers.

Out-of-order arrivals. Since the events sent from different nodes can arrive in any order, the algorithm for query evaluation needs to handle out-of-order arrivals. Our CEP engine achieves this by exploiting an evaluation model based on tree NFAs [16], where for each state, there are as many transitions into subsequent states as required event types that have not yet been processed up to that point. Moreover, all potential sequence constraints of a query and predicates must be satisfied and checked for a state transition.

Code Availability. The setup for all experiments is publicly available,[1] including instructions on how to reproduce the results.

7.2 Simulation Experiments

Network and Query Characteristics. We study the influence of network and query characteristics with a single query over 10 event types. Fig. 5 shows the transmission ratios obtained with PrePP plans constructed with the greedy and the sampling algorithm. In all experiments, the PrePP plans constructed with our sampling algorithm clearly outperform the greedy plan construction. Hence, we focus our discussion on the sampling-based plans.

Event node ratio. PrePP plans benefit most when the event node ratio is low, i.e., there are not many sources per event type, as in this case, only a few pull requests have to be sent in each step. For an event node ratio of 0.1, the sampling-based PrePP plan achieves a transmission ratio of 0.006 (Fig. 5a). Increasing the event node ratio increases the global rate of all event types in the network. Hence, pull requests become more expensive. However, even for an event node ratio of 1.0, the PrePP plans reduce event transmission by around two orders of magnitude.

Event skew. PrePP plans benefit from a high event skew: Events generated with low frequency are pushed to the sink node at low cost and then used for effective filtering when pulling the events of the remaining types. For an event skew of 1.1, the sampling-based PrePP plans achieve a transmission ratio of 1%(Fig. 5b). As the event skew decreases, the above effect decreases, and, therefore, the transmission ratio increases. The results of this experiment show the highest variance as for each data point new rates are drawn for each event type of the query by a Zipf random generator.

Number of nodes. The network size behaves similarly to the event node ratio, with the difference being that the number of sources per event type is not bounded. Thus, PrePP plans benefit most for small networks and achieve a transmission ratio of 0.007 for 10 nodes, see Fig. 5c. Yet, even for networks with 250 nodes, the sampling-based PrePP plan has a transmission ratio of only 0.07.

Minimal selectivity. The smaller the minimum selectivity incorporated in a query, the more selective it is, which leads to larger filtering opportunities when pulling events. For a minimal selectivity of 10^{-6}, the PrePP plan constructed with the sampling algorithm reduces transmission costs by three orders of magnitude.

Combination with In-network Processing. We compared the transmission costs of PrePP plans, MuSE graphs, and MuSE graphs extended with PrePP plans. We used a query workload comprising five queries, each containing at most six event types. We applied the exact algorithm for the computation of PrePP plans. Comparing the results obtained for MuSE graphs and Push-Pull MuSE graphs, the optimization potential is low in general(Fig. 6). Due to the decomposition of the query workload into many projections, the number of event types required to evaluate a projection decreases. However, PrePP plans can hardly optimize the event acquisition based on a few event types. Yet, we note some optimization potential in the experiment on the network size, see Fig. 6c. Increasing the network size leads to multi-sink placements being more costly since the number of nodes producing a particular event type increases. Therefore, more events are pushed in general, which yields optimization potential for push-pull-based communication.

PrePP Plan Construction. Next, we focus on constructing PrePP plans with sampling. To this end, Fig. 7 reports on the quality of the obtained plans in terms of the transmission ratio and the time required to construct them for different variations of this algorithm when varying the query length. The latter is captured by the number of event types to acquire for query evaluation.

Comparison to optimal plans. Fig. 7a illustrates that, as the number of event types to be acquired increases, so does the potential for optimization. For the sampling algorithm, we have set both the sample size s and the pruning parameter k to the number of event types to acquire. Despite the small number of samples, we note that the algorithm constructs near-optimal PrePP plans. In addition, we investigated a configuration of the sampling algorithm in which the top-k samples are selected as the best-k single-step PrePP plans before the partitioning phase of our algorithm is applied. This configuration stays very close to the optimal plan, which underlines our design choice to perform the partitioning solely based on a sample of the single-step plans.

[1] https://github.com/spurtzel/PrePP

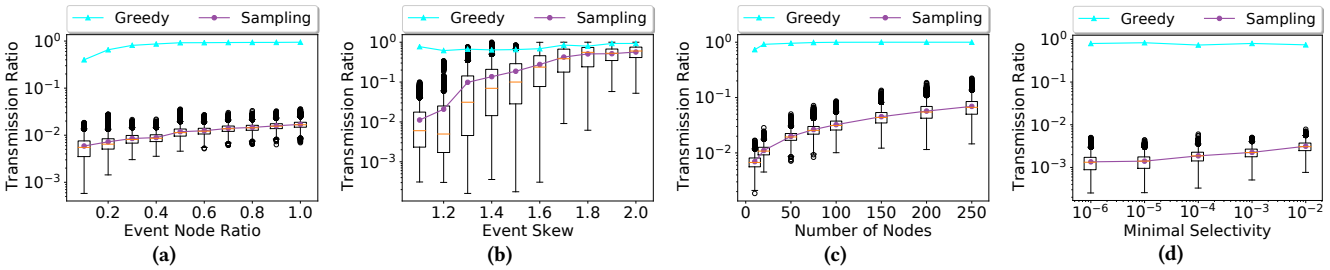

Figure 5: Varying network and query characteristics for a single query.

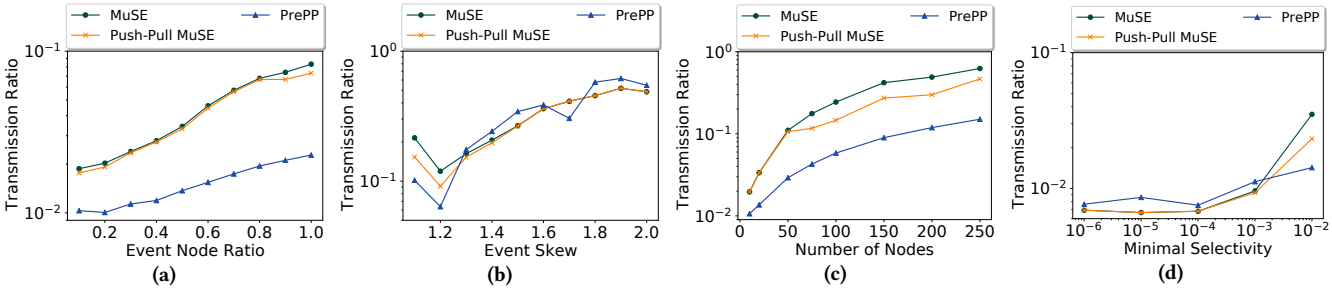

Figure 6: Varying network and query characteristics for multiple queries.

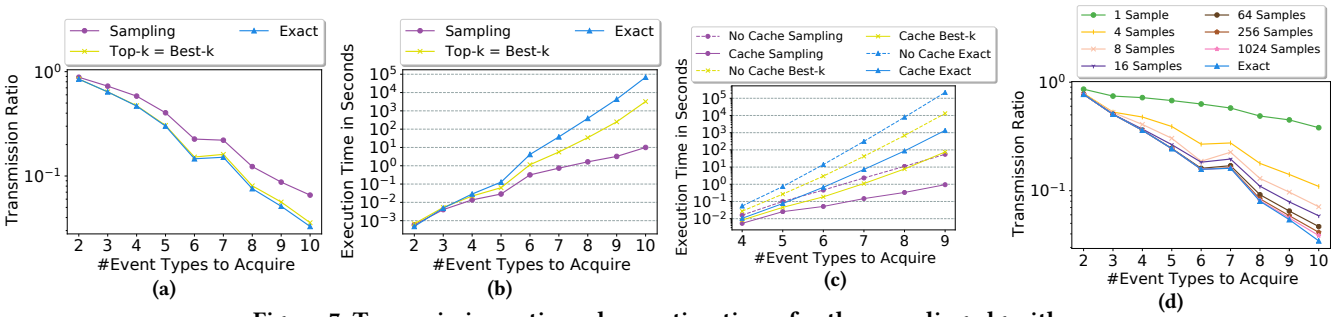

Figure 7: Transmission ratio and execution times for the sampling algorithm.

Next, Fig. 7b shows the times needed to construct the respective PrePP plans. As the number of types to be acquired increases, the runtime of all procedures also increases. Yet, for the sampling algorithm, the respective growth is much smaller compared to the other two algorithms. For queries consisting of 10 different event types, the sampling algorithm completes the construction of a plan in 10 seconds, which is practically feasible. The exact algorithm, in turn, needs around 10^5 seconds, i.e., 1.15 days, for plan construction.

Benefit of caching. The impact of our caching strategies is explored in Fig. 7c. We consider up to 9 event types to acquire, since for 10 types, the exact algorithm without caching did not terminate (timeout of four weeks). For all methods, it is apparent that caching is beneficial and reduces the runtime considerably. For 9 event types to acquire, we reduce the runtime by around two orders of magnitude for all algorithms. For the sampling algorithm, the benefits are even more pronounced, with runtime improvements of more than five orders of magnitude.

Impact of sample sizes. Fig. 7d shows the effect of increasing the number of samples s drawn in the construction of a PrePP plan. Clearly, a larger sample size improves the quality of the resulting plans. However, already for a relatively small sample of size 8, the result approximates the optimal plan very well. Also, for a larger number of samples, the resulting benefit becomes negligible.

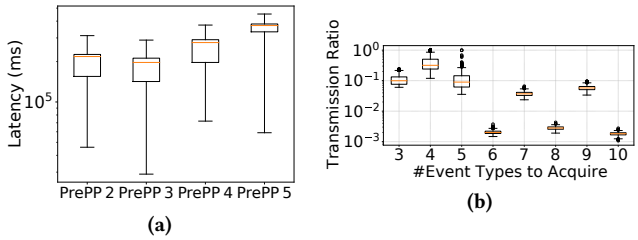

Figure 8: Detection latency and the impact of the selectivity.

Detection Latency. We also examined the effects of the length of PrePP plans on the detection latency. Fig. 8a shows that the detection latency increases with growing plan length. Such an increase is expected since the number of round-trips in event transmission increases due to pull-based communication. However, we observe that the detection latency for plans of length two, *PrePP 2*, is slightly higher than for plans of length three, *PrePP 3*. We attribute this to the processing latency induced by query evaluation. In this case, for a plan of length two, more events have to be sent over the network, which increases the number of events to process.

Impact of Selectivity. Fig. 8b illustrates the impact of the selectivities of the used queries on the transmission ratios obtained for the constructed PrePP plans. To this end, for different query

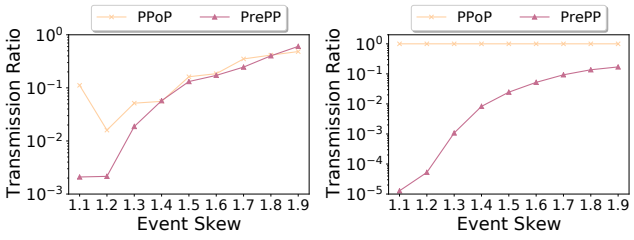

Figure 9: State-of-the-art comparison.

lengths, we introduced variance in the selectivity assigned to pairs of event types. The obtained transmission ratios are partially larger for shorter query lengths, since these plans includes less optimization potential through pull-based communication. However, in general, we see little variance in the transmission ratios over all query lengths, indicate a certain robustness of our approach.

Comparison with the State of the Art. Finally, we compared PrePP plans against a state-of-the-art technique for distributed CEP that adopts push-pull-based communication based on temporal constraints induced by a query time window [10]. Fig. 9a illustrates the results for networks with varying event skew, where all event types have rates lower than one. The latter is essential for pull-based approaches based on temporal constraints to be applicable. We observe that methods leverage large differences in the rates, but become less effective once the event skew increases. Fig. 9b, in turn, shows results for networks, in which all event types have a rate larger than one. As discussed earlier, pull requests based on time windows do not yield any benefit in these scenarios: The evaluation plan degrades and corresponds to a plan that would push all events. In contrast, our PrePP plans can still make effective use of pull-based communication due to the fine-granular filtering at event sources based on query predicates.

7.3 Case Study

Finally, we evaluated the efficiency of PrePP plans for distributed CEP in a case study with two real-world datasets.

Citi Bike. We used one month of the Citi Bike dataset [9]; a trace of 1 million events. Citi Bike is a bike-sharing service that publishes monthly information about completed bike rides. A ride is an event that consists of payload information such as the bike ID, trip duration, start/end station of a ride, and customer information. We defined 9 event types describing bike rides of varying duration, performed by customers of different age groups. We partitioned the events using the station ID to generate local traces for 20 nodes. Over the resulting event stream, we defined two CEP queries using a time window of 24 hours. Query 1 for the Citi Bike dataset in Listing 1 detects whether short, long, and very long trips with the same bike ID are made by premium and non-premium customers in an arbitrary order. From an application perspective, such a CEP query could provide analytical information about the frequency of a single bicycle being used within 24 hours, drawing attention to potential bike maintenance or indicating that particularly durable bikes might be worth renting to reduce the number of inspections. Based on the bike ID, we determined the selectivities of the query's predicates to generate PrePP plans. We evaluated the plans using our distributed CEP engine. The resulting transmission ratios can

```
Citi Bike Query 1:
PATTERN AND(ShortY sy, LongY ly, LongO lo, VLongC vc)
WHERE sy.bID=ly.bID ∧ ly.bID=lo.bID ∧ lo.bID = vc.bID
WITHIN 24h
Google Cluster Query 2:
PATTERN SEQ(Update1 u, Submit su, Schedule s, Finish
 f)
WHERE u.uID=su.uID ∧ su.uID=s.uID ∧ s.uID=f.uID
WITHIN 30min
```

Listing 1: Example queries used in the case study.

Table 3: Transmission ratios for case study.

	Query 1	Query 2
Citi Bike	0.002%	0.01%
Google Cluster	0.02%	0.008%

be found in Table 3. Due to the high selectivities, our PrePP plans resulted in a transmission ratio $\leq 0.01\%$ for both Citi Bike queries.

Google cluster. For the second case study, we used a Google cluster monitoring dataset [24], containing a trace with 4.35 million events. Attribute values include a machine/user/job ID or the priority for a particular job. There are 9 different event types representing the task life cycle. We partitioned the events using the machine ID to obtain local traces for 20 nodes. Again, we defined two CEP queries over the resulting event stream with a time window size of 30 minutes. Query 2 for the Google Cluster dataset in Listing 1 detects a sequence pattern, where a job was first updated, then submitted, afterward scheduled, and then finished. In practice, such a query helps to analyze the number of successful jobs fulfilling this sequence of actions for planning future processes. We derived the selectivities using the same job id for different tasks within the defined time window. The resulting PrePP plans generated a transmission ratio of $\leq 0.02\%$ for both queries, see Table 3.

Our case studies illlustrate that PrePP plans can successfully handle data distributions as encountered in real-world datasets.

8 RELATED WORK

CEP query evaluation shows exponential runtime in the number of events to be processed [25], i.e., a single event can double the number of partial matches to be maintained. As such, several strategies aiming for efficient query evaluation have been proposed, including sub-pattern sharing [19, 20], parallelization [6], data prefetching [27], lazy query evaluation [16], or load shedding [7, 26]. Kolchinsky et al. [16] introduced a lazy evaluation mechanism that exploits event rates and selectivities to reduce the number of partial matches to be maintained during the query evaluation. To this end, the evaluation automata are re-arranged in descending order of event rates so that events of the rarest event type are required to materialize a partial match. The underlying idea of PrePP plans is similar, as, dictated by the rates and selectivities, the acquisition order of a plan is chosen such that the number of partial matches generated and thereby, the number of messages sent is reduced. Zhao et al. [26] developed a hybrid load shedding model to shed input events and partial matches that will likely lead to no further match using the selectivities of query predicates. The resulting approach facilitates best-effort evaluation during overload situations. We adopt the idea of leveraging selectivities from predicates to reduce the number of partial matches, which in our case is reflected by the number of messages sent over the network.

Distributed CEP. Publish-subscribe systems realize some form of distributed event processing, where event producers initiate the communication (publish) and events are sent to subscribing consumers (subscribe). The PADRES system [13] denotes an instantiation of the pub-sub paradigm having a SQL-based declarative language (PSQL) for continuous queries over event streams with a time-based or count-based sliding window. To reduce communication costs of distributed event processing, operator placement strategies [3, 8, 18, 22], and frameworks [17] have been proposed. The idea is to decompose a query into its operators and assign them to network nodes for evaluation so that partial matches are exchanged between nodes to produce matches of the query. In our model, nodes also evaluate operators (projections). Yet, instead of partial matches, only the *atomic* events that create partial matches are sent over the network. Moreover, most operator placement approaches exclusively rely on push-based communication.

Push-pull plans. Push-pull-based CEP query evaluation in a network of event sources was first proposed by Akdere et al. [2] to prevent sending events ultimately not leading to matches. To this end, temporal constraints derived from the time window of a query are leveraged. Recently Flouris et al. [10] combined push-pull-plans as described in [2] with operator placement. However, as discussed, for time window-based pulling to be beneficial, some event types must have sufficiently low rates to not occur in each time window. We showed empirically that our PrePP plans outperform their approach. In [23], push-pull plans for queries over event streams created by dedicated processes are proposed that leverage behavioral constraints resulting from business process models, e.g., mutual exclusion or checks on IDs [23]. However, unlike our work, in [23], predicates are based on an underlying business process model, and the employed push-pull plans aim to reduce memory consumption.

9 CONCLUSIONS

We proposed PrePP plans as a model for evaluating CEP queries in networks. While we focused on a query language restricted to the operators *SEQ*, *AND*, and *OR*, support for negation and Kleene closure can naturally be incorporated into our PrePP model and is subject to future work. PrePP plans specify the acquisition order for event types and include pull sets, which enable fine-granular filtering by exchanging events to evaluate predicates. We further presented a cost model, characterized optimal PrePP plans, and proved NP-completeness for constructing an acquisition step, which renders the construction of optimal PrePP plans NP-hard. Striving for efficient plan construction, we developed a sampling algorithm and two caching strategies. They yield near-optimal PrePP plans while reducing the time for plan construction by up to five orders of magnitude over a naive algorithm. Our experiments also demonstrated that PrePP plans may reduce event transmission in query evaluation by up to three orders of magnitude.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG), project-ID 414984028, CRC 1404.

REFERENCES

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *SIGMOD*. ACM, 147–160.
[2] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. 2008. Plan-based complex event detection across distributed sources. *VLDB* 1, 1 (2008), 66–77.
[3] Samira Akili and Matthias Weidlich. 2021. MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks. In *SIGMOD*. ACM, 10–22.
[4] Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex Event Recognition Languages: Tutorial. In *DEBS*. ACM, 7–10.
[5] Ralph W. Bailey. 1998. The number of weak orderings of a finite set. *Social Choice and Welfare* 15, 4 (1998), 559–562.
[6] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. RIP: run-based intra-query parallelism for scalable complex event processing. In *DEBS*. ACM, 3–14.
[7] Koral Chapnik, Ilya Kolchinsky, and Assaf Schuster. 2021. DARLING: Data-Aware Load Shedding in Complex Event Processing Systems. *VLDB* 15, 3 (2021), 541–554.
[8] Jianxia Chen, Lakshmish Ramaswamy, David K Lowenthal, and Shivkumar Kalyanaraman. 2012. Comet: Decentralized complex event detection in mobile delay tolerant networks. In *MDM*. IEEE, 131–136.
[9] Citi Bike. 2022. Posted at http://www.citibikenyc.com/system-data.
[10] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Network-wide complex event processing over geographically distributed data sources. *Inf. Syst.* 88 (2020).
[11] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: A survey. *VLDB J.* 29, 1 (2020), 313–352.
[12] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *VLDB* 13, 5 (2020), 588–601.
[13] Hans-Arno Jacobsen, Vinod Muthusamy, and Guoli Li. 2009. The PADRES Event Processing Network: Uniform Querying of Past and Future Events. In *it - Information Technology*. 250–260.
[14] Mohit Jain, Vikas Chandan, Marilena Minou, George A. Thanos, Tri Kurniawan Wijaya, Achim Lindt, and Arne Gylling. 2015. Methodologies for effective demand response messaging. In *SmartGridComm*. IEEE, 453–458.
[15] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*. Springer, 85–103.
[16] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy evaluation methods for detecting complex events. In *DEBS*. ACM, 34–45.
[17] Manisha Luthra, Boris Koldehofe, Niels Danger, Pascal Weisenburger, Guido Salvaneschi, and Ioannis Stavrakakis. 2021. TCEP: Transitions in operator placement to adapt to dynamic network environments. *J. Comput. Syst. Sci.* 122 (2021), 94–125.
[18] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In ICDE. IEEE, 49.
[19] Olga Poppe, Chuan Lei, Lei Ma, Allison Rozet, and Elke A. Rundensteiner. 2021. To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams. In *SIGMOD*. ACM, 1452–1464.
[20] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *SIGMOD*. ACM, 495–510.
[21] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS*. ACM.
[22] Fabrice Starks and Thomas Peter Plagemann. 2015. Operator placement for efficient distributed complex event processing in MANETs. In WiMob. IEEE, 83–90.
[23] Matthias Weidlich, Holger Ziekow, Avigdor Gal, Jan Mendling, and Mathias Weske. 2014. Optimizing Event Pattern Matching Using Business Process Models. *IEEE TKDE* 26, 11 (2014), 2759–2773.
[24] John Wilkes. 2020. Yet more Google compute cluster trace data. https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.
[25] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*. ACM, 217–228.
[26] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load Shedding for Complex Event Processing: Input-based and State-based Techniques. In *ICDE*. IEEE, 1093–1104.
[27] Bo Zhao, Han van der Aa, Thanh Tam Nguyen, Quoc Viet Hung Nguyen, and Matthias Weidlich. 2021. EIRES: Efficient Integration of Remote Data in Event Stream Processing. In *SIGMOD*. ACM, 2128–2141.