DecoPa: Query Decomposition for Parallel Complex Event Processing

SAMIRA AKILI, Humboldt-Universität zu Berlin, Germany
STEVEN PURTZEL, Humboldt-Universität zu Berlin, Germany
MATTHIAS WEIDLICH, Humboldt-Universität zu Berlin, Germany

Systems for Complex Event Processing (CEP) enable the detection of predefined patterns in event streams. While the evaluation of CEP queries is computationally hard, scalability may be achieved by parallelization. Yet, existing approaches for parallel CEP are driven by static query properties, such as partitioning keys and states of the evaluation model. They largely neglect the rates with which processing units may ingest and compare events for query evaluation.

In this paper, we present an approach for parallel CEP that is based on a flexible decomposition of CEP queries. Our idea is to guide the decomposition by the sustainable throughput of each processing unit, in order to maximize the overall performance. To this end, we introduce DecoPa plans for parallel CEP, provide a cost model for them, elaborate on their correctness and optimality, and present an algorithm for their construction. Experiments using a DecoPa implementation in Flink illustrate throughput gains of up to 12 orders of magnitude compared to state-of-the-art approaches.

CCS Concepts: • **Information systems → Data streams**.

Additional Key Words and Phrases: Complex Event Processing, Parallel Processing, Stream Processing

ACM Reference Format:
Samira Akili, Steven Purtzel, and Matthias Weidlich. 2024. DecoPa: Query Decomposition for Parallel Complex Event Processing. *ACM Trans. Datab. Syst.* 2, 3 (SIGMOD), Article 132 (June 2024), 26 pages. https://doi.org/10.1145/3654935

1 INTRODUCTION

Complex Event Processing (CEP) is a computational paradigm that enables real-time pattern detection over streams of events [14] in domains like finance [26], healthcare [27], and transportation [4]. It enables the detection of *situations of interest* by matching CEP queries, which are characterized in terms of their event types, the ordering thereof, a set of predicates, and a time window [14].

Having low-latency processing of high-frequency event streams as its primary objective, a plethora of methods have been introduced to enhance the efficiency of CEP, including parallelization [5, 31], state sharing mechanisms [17, 22], and query rewriting [11, 28]. However, the evaluation of CEP queries is inherently difficult. When adopting a permissive query semantics, known as *unconstraint* [8] or *skip-till-any-match* [32], it may be necessary to compare all events of the types referenced in a CEP query that occur within the same time window in a stream against each other, in order to evaluate the query predicates. As such, common evaluation algorithms show an exponential worst-case time complexity, while the issue is even exacerbated once CEP queries contain Kleene Closure operators [32].

Authors' addresses: Samira Akili, Humboldt-Universität zu Berlin, Germany, akilsami@hu-berlin.de; Steven Purtzel, Humboldt-Universität zu Berlin, Germany, purtzest@informatik.hu-berlin.de; Matthias Weidlich, Humboldt-Universität zu Berlin, Germany, matthias.weidlich@hu-berlin.de.

ACM 0362-5915/2024/6-ART132
https://doi.org/10.1145/3654935

The efficiency of a CEP system is assessed in terms of its *sustainable throughput*, i.e, the maximal number of events that can be processed per time unit without generating back-pressure in terms of increased queueing of events [16]. For a CEP system, the sustainable throughput is dominated by two factors: the maximal number of events that can be ingested per time unit (ingestion rate ι) and the maximal number of comparisons between events, i.e., predicate evaluations, that can be made per time unit (comparison rate κ). Either rate is influenced by the semantics of CEP queries as well as the adopted CEP system. However, once measurements of both rates for a query q evaluated by a system s are available, they can be combined with information on the distribution of event rates and predicate selectivities to determine the sustainable throughput of s for q. In particular, if sustainable throughput cannot be reached at all, the rate with which events are fetched from event sources needs to be adjusted in order to prevent an overload situation.

Example 1. *Consider a system with a max ingestion rate $\iota = 4000e/min$ and a max comparison rate $\kappa = 100000/min$ for a CEP query $q = AND(A, B)$ that matches joint occurrences of events of types A and B that satisfy a set of predicates q_π and occur within a time window $q_t = 1min$. Let $r(A), r(B)$ denote the number of occurrences of A and B events per minute and $c \in (0, 1]$ be a scaling factor. Sustainable throughput requires that $\iota > c \cdot (r(A) + r(B))$. To evaluate the predicates q_π, each incoming A event is compared against B events within the time window, which results in $c \cdot r(A) \cdot r(B)$ comparisons. Considering also the comparisons for incoming B events, sustainable throughput requires that $\kappa > 2 \cdot r(A) \cdot r(B) \cdot c$.*

For the above setting, Fig. 1 shows the sustainable throughput for two event rate distributions $D1, D2$, for an increasing scaling factor. Here, the left and right y-axes describe the resulting ingestion rate and comparison rate, respectively, to achieve sustainable throughput, while the max ingestion and comparison rates ι and κ of the system are marked with dashed lines. The example illustrates that the required ingestion rate is independent of the event rate distribution, unlike the the comparison rate. Hence, depending on the event rate distribution, the sustainable throughput may be limited either by the max comparison rate, as for $D1$, or by the ingestion rate, as for $D2$.

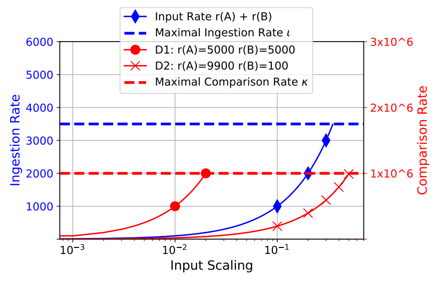


Fig. 1. Illustration of sustainable throughput.

In this paper, we argue that the interplay between ingestion rates and comparison rates shall guide the parallelization of CEP queries. Once a CEP system comprises multiple processing units, *each* of these units must achieve sustainable throughput. Yet, existing approaches for parallel CEP, whether data-parallel [15, 21], state-parallel [6], or using a hybrid model [30, 31], organize the distribution of the load based on static query properties, such as partitioning keys or the number of states of the evaluation model. As such, they neglect the ingestion rates and comparison rates of processing units and, therefore, cannot maximize the overall throughput.

Against this background, we present Decomposition-based Parallelization (DecoPa) of CEP, based on the following contributions:

- We introduce DecoPa plans as a model that encodes the decomposition of a CEP query and a partitioning of the event stream in order to facilitate parallel query evaluation.
- We provide a flexible cost model to reason about the (maximal) throughput that can be achieved for the evaluation of DecoPa plans in different processing contexts.
- We show the correctness of DecoPa plans and characterize their optimality in terms of throughput maximization.

- We present an algorithm for the construction of DecoPa plans, also lifting them to a multi-query setting.

We evaluate the parallel evaluation of CEP workloads based on DecoPa plans with an implementation based on the CEP library of Flink [7], using real-world and synthetic data sets. Our results show that DecoPa plans yield up to 12 orders of magnitude higher throughput compared to state-of-the-art approaches for parallel CEP. In the remainder, we first review related work and provide a motivating example (§2), before giving preliminaries (§3) and formalizing the problem setting (§4). Next, we introduce DecoPa plans, including a cost model, and elaborate on their properties (§5). Subsequently, we present an algorithm to construct DecoPa plans (§6). Finally, we present evaluation results (§7) and conclude (§8).

2 BACKGROUND

2.1 Related Work

Due to the complexity of CEP query evaluation [32], various optimization strategies have been proposed, including parallelization and query decomposition. While we build upon ideas for decomposition in CEP [11, 19, 28], distributed CEP [2, 3, 25], and multi-query optimization [17, 22, 23], our approach introduces a novel perspective. We focus on optimizing sustainable throughput in a parallel processing context, by guiding the decomposition by the processing capacity.

The three primary paradigms for parallel CEP are data-parallel, state-parallel, and parallel approaches [14, 24]. In [15], data-parallel CEP based on partitioning keys is proposed and cannot be applied when those are not given.Window-based approaches [20, 21] assign processing units to event stream partitions corresponding to different windows, characterized, e.g., by the query's time window or predicates. When using sliding time windows with significant overlap under a permissive selection policy, window-based parallelization can lead to substantial data replication. Run-intra-based parallelism (RIP) [5] is a form of data-parallelism where runs of an evaluation automaton are parallelized by sending overlapping batches of events to processing units. RIP was designed for contiguous event selection with a maximum match length and its adaption to selection strategies with unbounded match lengths requires batch sizes, which strain the processing unit's maximal ingestion rates.

State-based approaches split the CEP query's evaluation automaton into states and assign each state a single processing unit [6], which limits the degree of parallelism by the number of states.

Hybrid approaches combine state-parallelism with data-parallelism, further parallelizing the evaluation of each state by assigning its evaluation to a set of processing units. Variations of hybrid models consider shared-nothing or shared-memory settings [30, 31]. For the shared-nothing context, see [30], the input stream of each state, comprising primitive events from the global input stream and matches produced by previous states, is partitioned based on its expected input event. Hypersonic [31] adopts a similar approach in a shared-memory context. Processing units are divided into two groups: match-workers and event-workers. Each group processes either incoming matches from previous states or events from the primitive event stream, which reduces the required comparison costs. However, the approach is feasible only without extensive data replication, since all processing units can access all input events belonging to their state. We consider the more general, shared-nothing context. While we also employ the parallelization of states, i.e., sub-queries, we propose to decompose queries to meet processing constraints before assigning resources. Yet, in a shared-memory context, Hypersonic's parallelization strategy is orthogonal to our work, as it can be applied for the evaluation of each derived sub-query.

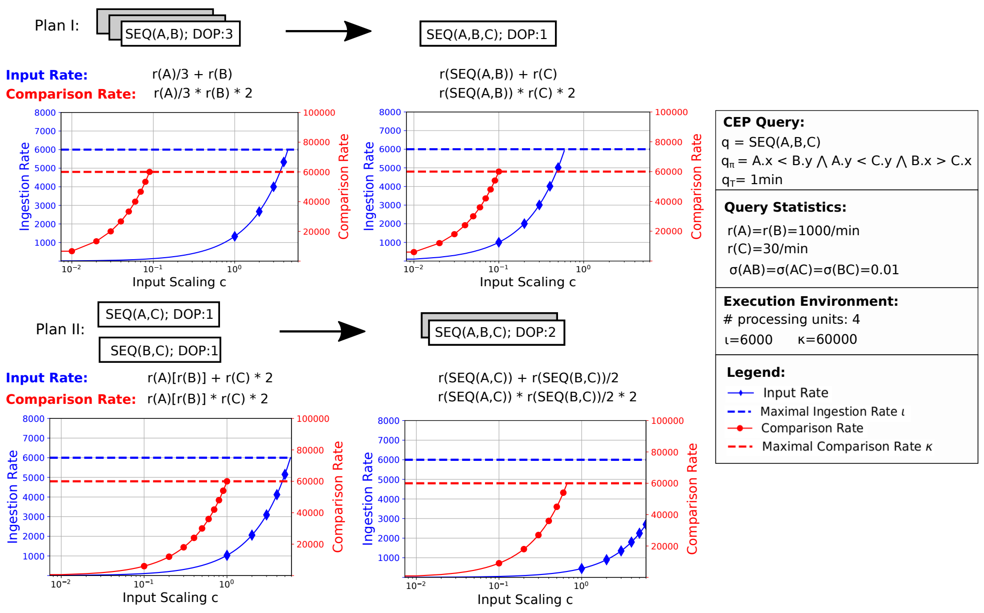


Fig. 2. Two plans for the parallel evaluation of the query q on 4 processing units.

2.2 Motivating Example

Fig. 2 illustrates two parallelization schemes for the evaluation of the CEP query $q = SEQ(A, B, C)$ defined over event types A, B, C. For a triple of events (A_i, B_j, C_k) to represent a match of q, they must fall into the time window $q_T = 1min$ and satisfy the predicates q_π, e.g, for the attributes of A_i, B_j it must hold $A_i.x < B_j.y$. The rates per event types are given as $r(A) = r(B) = 1000/min$ and $r(C) = 30/min$, while the selectivities of pairwise predicates in q_π are $\sigma(X) = 0.01$ with $X \in \{AB, AC, BC\}$. The query q is evaluated in parallel on a set of 4 processing units each having a max ingestion rate $\iota = 6000e/min$ and max comparison rate $\kappa = 60000e/min$.

Hybrid Parallelization. Plan I shows a naive decomposition of the query derived from the states of an evaluation automaton for q. The sub-query of the plan, $SEQ(A, B)$, is induced by the partial matches per state. As the sub-query is evaluated in parallel, the plan illustrates the main characteristics of the aforementioned hybrid approaches. The first step takes as input A and B events, to generate matches of sub-query $SEQ(A, B)$, which are then used to construct matches of $SEQ(A, B, C)$ by incorporating C events. $SEQ(A, B)$ is evaluated in parallel by 3 processing units, each receiving a third of the rate of the A's and all B's. With $r(A) = r(B) = 1000e/min$, for a scaling factor $c \in (0, 1]$, the number of events to ingest is $c \cdot (r(A)/3 + r(B))$ and the comparisons to be made are $c \cdot r(A)/3 \cdot r(B) \cdot 2$. The query q is evaluated based on $SEQ(A, B)$ matches and C events by a single processing unit ingesting $c \cdot (r(SEQ(A, B)) + r(C))$ events, and subsequently carrying out $c \cdot r(SEQ(A, B)) \cdot r(C) \cdot 2$ comparisons. The plots show, how the comparison rate and ingestion rate grow with c. As all processing units must achieve sustainable throughput, the first crossing of ι or κ marks the max throughput that can be achieved. For plan I, this is the case for κ at $x = 0.09$.

Decomposition-based Parallel. Plan II leverages query decomposition by deriving the matches of q based on the two sub-queries $SEQ(A, C)$ and $SEQ(B, C)$, with processing units of both sub-queries receiving the same C events to guarantee result completeness. Then, the number of events to ingest is $c \cdot (r(A)[r(B)] + r(C))$ and $c \cdot r(A)[r(B)] \cdot r(C) \cdot 2$ comparisons are needed for the processing unit matching $SEQ(A, C)$ and $SEQ(A, B)$, respectively. Two processing units are allocated to the matching of query q, yielding for each processing unit $c \cdot (r(SEQ(A, B))/2 + r(SEQ(A, C))$ events to

Table 1. Overview of notations for queries and splits.

Notation	Explanation
$e.id$, $e.t$, $e.type$,	Unique identifier, timestamp, and type of event
$\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_n\}$	Universe of event types
$q = (G_q, q_{pi}, q_t)$	Query with pattern tree G_q, predicates q_{pi}, and time window q_t
$G_q = (F, \varphi)$	Pattern tree with pattern functions F, complex pattern functions F_c, and primitive pattern functions F_p
$f.sem$	Semantic type of pattern function f of a pattern tree
$types(q)$	Event types referenced in query q
Q	Query workload
$Kleenetypes_q$, Ω_q	Child of KL pattern function, order constraints of query q
$s(q) = (S_q, E_q)$	Split of query q into DAG of sub-queries S_q

ingest and, analogously, to carry out $c \cdot r(SEQ(A, B))/2 \cdot r(SEQ(A, C) \cdot 2$ comparisons. Sustainable throughput can be reached up to a scaling factor of around $c = 0.65$ with the comparison rate for matching q being the limiting factor. Hence, decomposition-based parallelization achieves 7× higher throughput than the hybrid approach, leveraging the same number of processing units.

3 PRELIMINARIES

3.1 Query Language

Let $\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_n\}$ be a set of event types defining a schema as a set of attributes [14]. An *event* is an instantiation of an event type, i.e., a state change in a system, and is assigned a unique identifier $e.id$, an occurrence timestamp $e.t$, and its type $e.type \in \mathcal{E}$.

Syntax. A query $q = (G_q, q_\pi, q_t)$ comprises a pattern tree G_q, describing the event pattern required by the query, a set of predicates q_π, and time window q_t. We restrict our model to independent predicates that are defined over at most two event types referenced in the query.[1] The pattern tree $G_q = (F, \varphi)$ is an ordered tree, having pattern functions F as its vertices, which either refer to a single event type or encode some pattern for its child operators in the tree. We denote the former as primitive pattern functions F_p and the latter as complex pattern functions F_c. The function $\varphi : F \nrightarrow F^k$ with $k \in \mathcal{N}$ assigns to a pattern function a sequence of children. A pattern function has a semantic type, denoted as $f.sem$, which is for a complex pattern function one of $\{AND, SEQ, OR, KL\}$ and for a primitive pattern function given as event type in $\mathcal{E}$. For a query with the primitive pattern functions F_p in its pattern tree, we use $types(q) = \bigcup_{f \in F_p} f.sem$ to denote the event types referenced in the query q. Moreover, by $\#_q(\epsilon)$, we denote the number of primitive pattern functions of q having the semantic type ϵ. We restrict our query model to queries in which the pattern functions f with $f.sem = KL$ have exactly one child given as a primitive pattern function. Moreover, to simplify notations, in our query model, Kleene Closure cannot be defined multiple times over the same event type. A query workload Q, is a set of queries. We split any query containing an OR pattern function into a query workload, so this does not constrain our model's expressiveness.

Semantics. To define the semantics of a query $q = (G_q, q_\pi, q_t)$, we introduce the set of *order constraints* Ω_q and the $Kleenetypes_q$, which can both be derived from the query tree. Here, $Kleenetypes_q = \{f_i.sem \mid f_i \in \varphi(f) \wedge f \in F \wedge f.sem = KL\}$ are the semantic types of the primitive pattern functions that are children of a pattern function of type KL. For the definition of the order constraints, let $\phi(f)$ denote the leaf nodes in a pattern tree that are reachable from the pattern function f, and $root(q)$ denote the root pattern function of G_q. The set of order constraints for a pattern function f is:

[1]This restriction will be discussed in more detail later.

$$\Omega(f) = \begin{cases} \bigcup_{i \in \varphi(i)} \Omega(i) \cup \bigcup_{(k_i,k_j) \in \{(x_i,y_j) | x_i, y_j \in \varphi(f) \wedge i<j\}} \phi(k_i) \times \phi(k_j) & \text{if } f.sem = SEQ \\ \bigcup_{i \in \varphi(i)} \Omega(i) & \text{otherwise.} \end{cases}$$

The order constraints Ω_q of a query q are given by $\Omega(root(q))$.

The semantics of a query is defined over a stream of events $s = \{e_1, e_2, \ldots\}$. Let $q = ((F, \varphi), q_\pi, q_t)$ be a query with $Kleenetypes_q$ and order constraints Ω_q. Moreover, let $M : F_p \longrightarrow 2^s$ be an injective function, mapping the primitive pattern functions F_p to subsets of events in s. Note that only for the primitive pattern functions corresponding to $Kleenetypes_q$, the respective subsets of events comprise more than one element. Then, $m = \cup_{o \in F_p} M(o)$ is a match of q if:

(i.) all events in m occur within the time window q_t and fulfill the pairwise predicates q_π;
(ii.) operators are mapped to events having the same type, $\forall o_i \in F_p, M(o_i) = E_i : \forall e_i \in E_i : o_i.sem = e_i.type$;
(iii.) events of the mapping fulfill the order constraints of the respective operators, $\forall (o_i, o_j) \in \Omega_q, M(o_i) = E_i, M(o_j) = E_j : \forall e_i \in E_i, e_j \in E_j : e_i.t < e_j.t$; and
(iv.) the number of occurrences of events per type equals the number of operators of this type in the query, $|\{e \mid e \in m \wedge e.type \in Kleenetypes_q\}| \geq \#_q(\epsilon)$ and $\forall \epsilon \in types(q) \setminus Kleenetypes_q : |\{e \mid e \in m \wedge e.type = \epsilon\}| = \#_q(\epsilon)$.

The above corresponds to the least restrictive matching semantics, known as *unconstrained* [8] or *skip-till-any-match* [1] which does not constrain the number of times an event is part of a match.

3.2 Partial Match based Query Evaluation

We assume a partial match-based query evaluation mechanism, as it is realized by NFA-based CEP evaluation and implemented in common stream processing engines [7, 12]. That is, matches of a query q are constructed incrementally by extending *partial matches* denoting matches of sub-queries of q, with the sub-queries referring to the states of an evaluation automaton. In Fig. 3, an evaluation automaton for the query $q = SEQ(A, B, AND(C, D))$ is illustrated.[2] Each state refers to a sub-query for which matches are generated, e.g., partial matches of state 3 refer to matches of the sub-query $SEQ(A, B, D)$. In the example, the partial matches for each state (matches for each sub-query) are shown after evaluating q over the stream s up to the event $C2$. Let $query(k)$ denote the sub-query of q evaluated at state k. An arriving event e_i causes the extension of a partial match $pm = \{e_{j1}, \ldots, e_{jn}\}$ of state k to a partial match of state $k + 1$ if $e_i \cup pm$ yields a match of the query $query(k + 1)$.

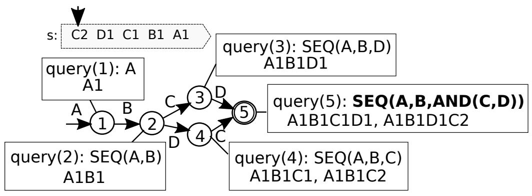


Fig. 3. Partial match based query evaluation of q.

3.3 System Model

For the parallel evaluation, a CEP query workload is processed in a shared-nothing cluster comprising the processing units P. A processing unit is a conceptual notion and can be mapped to, e.g., a core, a thread, or an operator in a distributed stream processing plan. It refers to exactly one instance of a sub-query and allows fine-granular construction of parallelization plans.

[2]For illustration purposes, q is only defined in terms of its pattern.

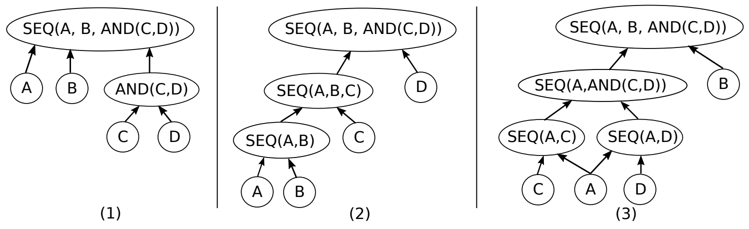


Fig. 4. Different query splits.

The processing units employ a partial match-based CEP query evaluation mechanism and are homogeneous in terms of their processing capabilities, i.e., they have an maximal ingestion rate ι, and a maximal comparison rate κ for a given set of query predicates. While the comparison rate depends on the complexity of the query predicates, the ingestion rate is not only influenced by workload properties. It limits the ability to take in input events and depends on the characteristics of the adopted CEP system. As such, it is affected by, e.g., CPU cycles per second, the time needed to maintain data structures per event, and potentially garbage collection.

The processing units may exchange events directly, i.e., the system topology is a complete graph. The event stream, over which a workload is evaluated, is distributed to a set of processing units using a shuffling process, based on a parallelization plan.

4 PROBLEM SETTING

4.1 Query Splits

A query split function, denoted as $s : q \longrightarrow S_q \times E_q$, decomposes a CEP query q into a directed acyclic graph (DAG) having sub-queries S_q of q as vertices. Fig. 4 illustrates three different query splits for $SEQ(A, B, AND(C, D))$ as they have been proposed for distributed and parallel CEP query evaluation. (1) shows a syntactic query split, as proposed for operator placement in [13]. That is, sub-queries are derived from the complex pattern functions of the pattern tree of a query. (2) shows a split as it is used for state-based (and hybrid) parallelization approaches [30, 31]. The split is derived from the order of states of an evaluation automaton processing the query in the order of the leaf nodes of its pattern tree. As such, the sub-queries of the split are defined by queries for which (partial) matches are constructed in each state of a respective automaton. We denote such a split as left-deep (*ld*) split of a query. (3) shows a split that decomposes the query into sub-queries over arbitrary subsets of the event types in *types*(q) proposed for in-network evaluation in [2, 3].

We consider sub-queries defined over arbitrary event types of a query for the instantiation of the query split function. Though, to use the matches of a sub-query for generating matches of a query, we introduce the notion of *valid sub-queries.*

Definition 1 (Valid Sub-query). *Let $q = (G_q, q_\pi, q_t)$ be a query having $Kleenetypes_q$ and order constraints Ω_q, and $o = (G_o, o_\pi, o_t)$ be a query with $Kleenetypes_o$ and Ω_o. The query o is a valid sub-query of q, if:*

i. it has an equal time window, $o_t = q_t$;
ii. it contains only event types, predicates, Kleene types, and order constraints of q, $types(o) \subseteq types(q)$, $o_\pi \subseteq q_\pi$, $\Omega_o \subseteq \Omega_q$, and $Kleenetypes_o \subseteq Kleenetypes_q$;
iii. for each contained event type it contains all *respective operators of q referencing the type, $\forall \epsilon \in types(o) : \#_o(\epsilon) = \#_q(\epsilon)$.*

A valid query split enables correct and complete query evaluation . Let s denote a sequence of events and let $m(q, s)$ be the set of all matches of a query q that can be found in s. Also, let $pred(q_s)$ be the set of predecessor sub-queries of q_s in a split DAG. Then, for the split $s(q) = (S_q, E_q)$ of query

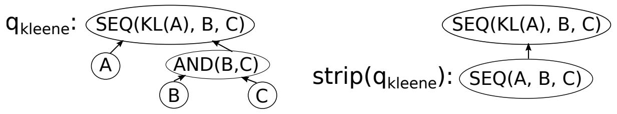


Fig. 5. Binary (left) and unary (right) Kleene split for q_{kleene}.

q to enable correct and complete query matching, it must be possible to construct each match of each sub-query $o \in S_q$ based on matches of preceding sub-queries, i.e., $m(o,s) \subset \bigtimes_{i \in pred(o)} \{\bigcup x \mid x \in \mathcal{P}(m(i,s))\}$. To this end, we consider *valid splits.*

Definition 2 (Valid Splits). *Let $s(q) = (S_q, E_q)$ be a split of the query q. The split $s(q)$ is valid if the following holds for each sub-query $o \in S_q \cup \{q\}$:*

i. $pred(o)$ are valid sub-queries of o
ii. the event types of the preceding sub-queries cover the event types of o, $\cup_{i \in pred(o)} types(i) = types(o)$

Property 1. *Valid splits enable correct and complete query matching.*

In the remainder, we focus on *binary splits*, in which each sub-query has at most two predecessors in the split DAG. We empirically observed that binary splits were mostly chosen by our construction algorithm aiming to minimize comparisons, which is explained as follows: When considering an evaluation automaton that can process events of each type immediately upon arrival, without buffering, the number of states grows exponentially with the number of inputs [18]. Since each event must be compared against the partial matches of nearly every state, the count of comparisons tends to be higher than with a series of automatons, each with only two inputs. As the number of comparisons depends on the number of partial matches, which is influenced by event rates and selectivities, the trend cannot be established for the general case, though.

Kleene Splits. Kleene Closure is matched for an event type ϵ of a query q if ϵ is a Kleene type of q and not a Kleene type of any of q's preceding sub-queries, i.e., $\exists \epsilon \in KleeneTypes_q : \forall o \in pred(q) : \neg \epsilon \in KleeneTypes_o$. We restrict ourselves to splits, in which per sub-query for at most one event type, Kleene Closure can be matched, and consider two types of splits for matching Kleene Closure: Given the split $s(q) = (S_q, E_q)$ for query q, let $q_{kleene} \in S_q$ denote a sub-query that comprises a pattern function f in its pattern tree with $f.sem = KL$, and let none of q_{kleene}'s predecessors in the split $s(q)$ comprise a Kleene Closure in its pattern tree. Moreover, let $\epsilon_{kleene} \in Kleenetypes_q$.

- In a *binary split*, q_{kleene} has two preceding sub-queries o_i, o_j whereas o_i is given as the (primitive query) having only one pattern function f_i in its query tree with $f_i.sem = \epsilon_{kleene}$ and o_j as a query that can be obtained from q_{kleene} by removing the pattern functions f_{j1}, f_{j2} with $f_{j1}.sem = \epsilon_{kleene}$ and $f_{j2}.sem = KL$ from q_{kleene}'s pattern tree.
- In a *unary split*, q_{kleene} has only one preceding sub-query, $strip(q_{kleene})$, which can be obtained from q_{kleene} by removing the pattern function f with $f.sem = KL$ from the pattern tree of q_{kleene}.

Fig. 5 shows a binary and unary split for the query $SEQ(KL(A), B, C)$. The matching complexity of the evaluation of queries with Kleene Closure based on binary and unary splits differs and is critical for the design of parallelization plans, which we will discuss later.

4.2 Problem Formulation

Given a query q defined over the event types $\mathcal{E}(q)$ and let $D : \mathcal{E}(q) \rightarrow \mathcal{R}$ assign to each event type its occurrences per time window q_t of the query. The query q shall be evaluated by a set P of homogeneous processing units for which ι denotes the maximal number of events that can be ingested per time window q_t. For the evaluation of the predicates q_π, let κ be the maximal number

Table 2. Notations for DecoPa plans.

Notation	Explanation
ι, κ	Maximal ingestion and comparison rate for a query and processing unit
$o = (o_q, o_{pe}, o_{part})$	Operator o with respective query, processing units and partitioning type
$PG = (O, E)$	Parallelization graph with operators O and edges E
$pre(o)$, $suc(o)$	Preceding and succeeding operators of o with respect to parallelization graph
$r(q)$	Occurrence rate of a query q
$\sigma(q)$	Selectivity induced by the predicates q_π of query q
$\hat{\iota_o}$, $\hat{\kappa_o}$	Ingestion and comparison costs of an operator o

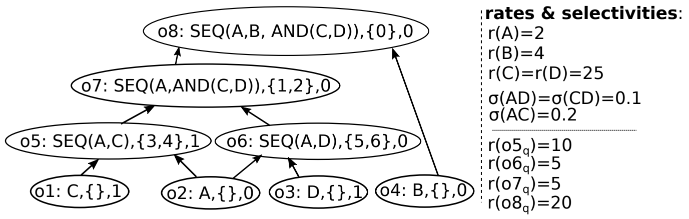


Fig. 6. Parallelization plan for query $SEQ(A, B, AND(C, D))$ on 7 processing units, along with output rates and selectivities of the comprised event types and sub-queries.

of comparisons that a processing unit can realize per time window. Decomposition-based parallel query evaluation leverages query splits to decompose q into a set of sub-queries S_q which are then processed by the processing units. A parallelization plan $\mathcal{P}$ describes the split and resource assignment. Evaluating a parallelization plan $\mathcal{P}$, such that κ and ι are respected, is only possible up to a certain value of the parameter c which denotes the fraction of the occurrences D of the event types to be processed per time window by a processing unit.

Example 2. *In Fig. 2, two parallelization plans for evaluating the query q on 4 processing units are shown. The split and resource assignment of plan 1 enable sustainable throughput for a maximal scaling value of $c = 0.09$, and for plan 2 of $c = 0.65$.*

Problem 1 (Efficient Parallel CEP Evaluation). *For a query and set of processing units with κ, ι, the problem of efficient parallel CEP evaluation is to construct a parallelization plan $\mathcal{P}$ that respects κ and ι, such that c is maximal.*

5 DECOMPOSITION-BASED PARALLELIZATION

We propose Decomposition-based Parallel (DecoPa) plans to address the problem of efficient, parallel CEP. We start with their formal definition (§5.1), before turning to their execution (§5.2). Then, we propose a cost model (§5.3), and investigate optimal DecoPa plans (§5.4). Finally, we lift DecoPa plans to query workloads and elaborate on plan adaptivity (§5.5).

5.1 DecoPa Model

A decomposition-based parallel (DecoPa) plan comprises the information necessary to enable parallel evaluation of a given CEP query on a set of processing units. Moreover, it encodes the partitioning scheme of the input event stream which is to be instantiated by a shuffling process to distribute incoming events to processing units.

We leverage binary query splits to achieve throughput optimizing DecoPa plans. That is, a query is decomposed into a graph having sub-queries as vertices and each sub-query having at most two predecessors. To evaluate the query, the sub-queries are evaluated in parallel by processing units.

The basic building block of a DecoPa plan is an *operator* which specifies a sub-query, a set of processing units to evaluate the sub-query on, and a flag denoting if matches of the sub-query shall be broadcasted or partitioned among the processing units.

Definition 3 (Operator). *Let Q be a set of queries and P a set of processing units. An operator is a triple $(Q, 2^P, \{0, 1\})$.*

For an operator $o = (o_q, o_{pu}, o_{part})$, o_q denotes the sub-query of the operator, o_{pu} are the processing units, and o_{part} is the partitioning type, i.e., if o_{part} is set to 0, matches of o_q are broadcasted, otherwise they are partitioned.

Definition 4 (DecoPa Plan). *Given a set of operators V_O. A DecoPa plan is a directed acyclic graph $PG = (V_O, E_O)$ having V_O as vertices and E_O as edges.*

Given a DecoPa plan $PG = (V_O, E_O)$ and operator $o \in V_O$, we denote $pre(o) = \{o_1 \mid (o_1, o_2) \in E_O\}$ as the preceding and $suc(o) = \{o_2 \mid (o_1, o_2) \in E_O\}$ and succeeding operators of o in PG.

Example 3. *Fig. 6 illustrates a parallelization plan for the query $q = SEQ(A, B, AND(C, D))$ evaluated on the processing units $\{0, \ldots, 6\}$, having the operators $\{o1, \ldots, o8\}$ as vertices. At the operator $o5$ the query $o_q = SEQ(A, C)$ is evaluated by the processing units $o_{pu} = \{3, 4\}$ and as $o_{part} = 1$ its matches are partitioned. For $o7$, $pre(o7) = \{o5, o6\}$ and $suc(o7) = \{o8\}$.*

5.2 Execution of DecoPa Plans

To execute a DecoPa plan $PG = (V_O, V_E)$ each operator $o \in V_O$ is instantiated by the set of processing units o_{pu}. The leaf operators of a parallelization plan stand for the primitive event types in the input stream and describe if instances thereof are partitioned among or broadcasted to their succeeding operators in PG. Based on the leaf operators, a shuffling process is instantiated. Let e be an instance of event type ϵ and o^{leaf} an operator in PG for which $o_q^{leaf} = \epsilon$. If $o_{part}^{leaf} = 0$, e is forwarded to all processing units of the suceeding operators $suc(o^{leaf})$. Otherwise, i.e., $o_{part}^{leaf} = 1$, for each operator $o^{suc} \in suc(o^{leaf})$ instances of ϵ are sent in a round-robin manner to the processing units in o_{pu}^{suc}.

A processing unit $u \in o_{pu}$ of the operator $o \in V_O$ matches the sub-query o_q based on matches of the sub-queries $\{o_q^p \mid o^p \in pre(o)\}$. To this end, u realizes partial match based query evaluation and compares each incoming match to its locally maintained partial matches, to form new matches of o_q. As for the leaf operators, if $o_{part} = 0$, each match of the sub-query o_q generated by the instance u is broadcasted to the processing units of the successors of o. If $o_{part} = 1$, for each operator $o^s \in suc(o)$ partitions of the matches of sub-query o_q^s are distributed among its processing units o_{pu}^s. The partitioning scheme employed for distributing the matches of o_q depends on the matching semantic of each succeeding operator of o:

1. *(binary split)* For a succeeding operator o^s matching the query o_q^s with a binary split, matches of o_q can be distributed in a round-robin manner among the processing units o_{pu}^s.
2. *(unary split)* If o is the (only) preceding operator of an operator o^{KL} which generates matches of Kleene Closure based on a unary split, then $o_q = strip(o^{KL})$ (§4.1). Let ϵ be the event type for which Kleene Closure is matched at o^{KL}. To generate all matches of o_q^{KL}, each match of o_q that is identical in its events of the types in $types(q) \setminus \epsilon$ needs to be processed by the same processing unit of o^{KL}. To this end, a hash-based partitioning on the IDs of the events of type $types(q) \setminus \epsilon$ is applied at the processing units of o_q.

Example 4. *For the evaluation of Fig. 6, a shuffler distributes events of type C among the processing units of operator $o5$, i.e., events of type C are forwarded alternately to the processing units $\{3, 4\}$. Analogously, D events are distributed among the processing units of $o6$. All A events are forwarded to* *all processing units of the operators* o5, o6 *and all B events to the processing unit of* o8*. Based on its respective share of C events and A events, each processing unit of* o5 *generates matches of the query SEQ(A, C) and forwards the matches alternately to the processing units of* o7*. Each processing unit of* o6 *forwards all matches to the processing units of* o7*. The processing units of* o7 *generate matches of SEQ(A, AND(C, D)) based on matches of SEQ(A, C) and SEQ(A, D) and forward all respective matches to the processing unit* 0 *of* o8 *which generates matches of the query q.*

5.3 Cost Model

5.3.1 Output Rates of Queries. To reason about the ingestion and comparisons to be made by a processing instance of an operator, we capture the rates with which matches of (sub)-queries are generated based on known distributions of event type occurrences and predicate selectivities. In order to calculate the selectivities of arbitrary sub-queries, we assume binary predicates that are independent. However, this restriction is unnecessary if the selectivities of all possible sub-queries of a given query are available. Let $(\epsilon_1, \epsilon_2) \in types(q)^2$ be a pair of primitive event types referenced in the query q, then $\sigma(\epsilon_1, \epsilon_2)$ denotes the selectivity of all predicates in q_π comprising ϵ_1, ϵ_2. We overload notation and define the selectivity of a query q which *does not contain Kleene Closure* as follows:

$$\sigma(q) = \prod_{(\epsilon_1,\epsilon_2)\in\{(x,y)|x,y\in \text{types}(q)\wedge x\neq y\}} \sigma(\epsilon_1, \epsilon_2)^{\#_q(\epsilon_1)\cdot\#_q(\epsilon_2)}. \prod_{(\epsilon_1,\epsilon_1)\in \text{types}(q)^2} \sigma(\epsilon_1, \epsilon_1)^{\frac{\#_q(\epsilon_1)^2}{2}}.$$

The first factor denotes the number of times predicates defined over events of different types are applied. Predicates defined over events of the same type referenced n times in the query are checked for each 2-element subset of the respective n events, constituting the second factor.

Example 5. *The selectivity of the query* $o7_q$ *in Fig. 6 is given by the product of the selectivities of the predicates defined over the tuples* $(A, C), (A, D), (C, D)$*, i.e.,* $\sigma(o7_q) = 0.1 \cdot 0.1 \cdot 0.2 = 0.002$.

Given an event type universe $\mathcal{E}$, let $r : \mathcal{E} \longrightarrow \mathcal{R}$ assign to the event types in $\mathcal{E}$ their number of occurrences per time unit u. Given the query $q = (G_q, q_\pi, q_t)$, we assume for simplification, that the occurrences r of event types in $types(q)$ are given with respect to the time window q_t, i.e., $u = q_t$.

The rate of a query is inductively defined along the pattern functions of the queries pattern tree $G_q = (F, \varphi)$. The rate of a primitive pattern function $f_p.sem \in F$ with semantic type $f_p = \epsilon$, is given by $r(\epsilon)$. For a complex pattern function $f_c \in F$, with $\varphi(f) = \langle f_1, \ldots, f_k\rangle$ the rate is defined inductively:

$$r(f) = \begin{cases} k \cdot \prod_{1\leq i\leq k} r(f_i) & \text{if } f.sem = AND, \\ \prod_{1\leq i\leq k} r(f_i) & \text{if } f.sem = SEQ, \end{cases}$$

Let $root(G_q)$ be the root pattern function of the pattern tree of the query q, then the rate of q is defined as $r(q) = \sigma(q) \cdot r(root(G_q))$.

Example 6. *The output rate of the query* $o7_q$ *in Fig. 6 is given by* $r(o7_q) = r(A) \cdot (r(C) \cdot r(D) \cdot 2) \cdot \sigma(o7_q) = 5$.

Output Rate of Kleene Closure Queries. To define the output rate of a query containing Kleene Closure q_{KL}, we first define the query q^-_{KL} that is obtained by removing all pattern functions having the semantic type KL and their child pattern functions corresponding to the $Kleenetypes_{q_{KL}}$ from the pattern tree of q_{KL}. More precisely, to transform q_{KL} to q^-_{KL}, each vertex k with $k.sem = KL$ as well as its child vertex is removed from the pattern tree of q_{KL}. Now, we can define the rate of the query q_{KL} as follows: Let $F_{kleene} = \{f_i \mid f_i \in \varphi(f) \wedge f.sem = KL\}$ be the set of primitive pattern functions of q_{KL}, which are child operators of a pattern function having KL as its semantic type. Then, the rate is defined as:

$$r(q_{KL}) = r(q_{KL}^{-}) \cdot \prod_{\epsilon_{KL}=x.sem \in F_{kleene}} \sum_{i=0}^{r(\epsilon_{KL})} \left(\binom{r(\epsilon_{KL})}{i} \cdot \sigma_{(\epsilon_{KL},\epsilon_{KL})}^{i-1} \cdot \prod_{x \in types(q_{KL}) \setminus \{\epsilon_{KL}\}} \sigma(x, \epsilon_{KL})^{i \cdot \#_{q_{KL}}(x)} \right).$$

The first factor in the equation indicates the rate of the sub-matches that can be part of a match of q_{KL} and do not contain Kleene sequences of the $Kleenetypes_{q_{KL}}$. Therefore, the rate is given by $r(q_{KL}^{-})$, which can be calculated as presented in the previous section. Each such sub-match can be combined with the matches of the Kleene operators F_{kleene} which comprise for each operator sequences of the respective ϵ_{kleene} events for which the predicates defined over pairs of ϵ_{kleene} events and for pairs of ϵ_{kleene} and events of types in $types(q_{KL}^{-})$ hold.

5.3.2 Costs of Operators. For a DecoPa plan $PG = (V_O, E_O)$, each processing unit of an operator $o \in V_O$, generates matches of the query o_q based on the matches generated by its preceding operators $pre(o)$. Let c denote the fraction of event type rates evaluated.

Input Costs. The input costs $\hat{\iota_o}$ of an operator are defined as:

$$\hat{\iota_o} = \sum_{o^{pre} \in pre(o)} c \cdot r(o_q^{pre}) \cdot \frac{1}{(|o_{pu}| - 1) \cdot o_{part}^{pre} + 1}.$$

For each predecessor operator o^{pre} for which matches are partitioned, i.e., $o_{part}^{pre} = 1$, each processing unit o_{pu} receives an equivalent share[3] of the matches of o_q^{pre}. Otherwise, each processing unit o_{pu} receives all matches of o_q^{pre}.

Comparison Costs. The comparison costs $\hat{\kappa_o}$ of an operator o capture the number of comparisons between the events received from preceding operators and the partial matches maintained by a processing unit of o per time unit. It depends on the number of matches generated per preceding operator, the classes of maintained partial matches, and the number of partial matches per class. The classes of partial matches, in turn, depend on the semantics of the query o_q of the operator o. In our model, we assume that the comparisons conducted within the context of each sub-query demand uniform time, implying that all predicate checks exhibit equal complexity. Should this assumption not hold, we propose constructing a plan that uses the maximum time required for predicate checks in any sub-query to guide the construction. We aim to explore a weighted model in future work to enable more fine-granular control over the comparison costs.

SEQ, AND. Let o_q be a query comprising only *SEQ* and *AND* pattern functions in its pattern tree and let $o1, o2 \in pred(o)$ denote the preceding operators of o. Each processing unit evaluating o_q maintains two classes of partial matches, each comprising its received partitions of all matches of $o1_q, o2_q$ respectively, that can be found per time window of q. The concrete number of instances in each class of partial matches equals the share of matches each processing unit receives from its predecessors, i.e., $r(o1_q)/|o_{pu}|$ if $o1_{part} = 1$, $r(o1_q)$ otherwise; analogously for o_2.

To match o_q, each incoming match of $o1_q$ is compared against all partial matches of type $o2_q$, and vice versa, resulting in comparison costs:

$$\hat{\kappa_o} = c \cdot 2 \cdot \frac{r(o1_q)}{(|o_{pu}| - 1) \cdot o1_{part} + 1} \cdot \frac{r(o2_q)}{(|o_{pu}| - 1) \cdot o2_{part} + 1}.$$

Kleene Comparisons. An operator o^{KL} matching Kleene Closure is an operator for which the respective query o_q^{KL} comprises Kleene Closure for *one* primitive event type ϵ for which none of

[3]While this assumption holds for non-Kleene matching, for unary Kleene splits, due to the hash-based partitioning scheme, processing units can receive varying shares of matches. Those may differ in their size due to skewed attribute values. However, this is not considered in our cost model.

the queries of the predecessors of o^{KL} comprise Kleene Closure. We distinguish between a unary and a binary split for decomposing the evaluation of Kleene Closure.

For a unary split, matches of o_q^{KL} are generated based on matches of $strip(o_q^{KL})$ (see §4). A processing unit generating o_q^{KL} matches from $strip(o_q^{KL})$ has to maintain only one class of partial matches which comprises valid matches of o_q^{KL}: An incoming match of $strip(o_q^{KL})$ already represents a match of o_q^{KL} and as such instantiates a partial match of the class. Moreover, partial matches of type o_q^{KL} can be extended to further matches of o_q^{KL} after processing incoming matches of $strip(o_q^{KL})$. Hence, the number of partial matches of type o_q^{KL} is given by $r(o_q^{KL})$. With o^{strip} as the preceding operator of o^{KL} matching $strip(o_q^{KL})$, the comparison rate of o^{KL} matched with a unary split is:

$$\kappa_{\hat{o^{KL}}} = c \cdot \frac{1}{(|o_{pu}^{KL}| - 1) \cdot o_{part}^{strip} + 1} \cdot r(strip(o_q^{KL})) \cdot r(o_q^{KL}).$$

Let $\epsilon_{KL} \in Kleenetypes_{KL_q}$ and o^{pre-KL} denote the preceding operator comprising ϵ_{KL} and $o^{pre-noKL}$ the second preceding operator. Moreover, let q_{KL} denote a query that is obtained by inserting the Kleene Closure pattern function into the pattern tree of o_q^{pre-KL} as parent node of the respective pattern function of ϵ_{KL}. To evaluate o_q^{KL} based on o_q^{pre-KL} and $o_q^{pre-noKL}$, three classes of partial matches need to be maintained per processing unit of o^{KL}. One class comprises matches of type $o_q^{pre-noKL}$ which are instantiated whenever a match of $o_q^{pre-noKL}$ is received. The second class comprises matches of q_{KL}, which are derived from received matches of o_{pre-KL}. The third class contains matches of the operators query o_q^{KL} which need to be maintained as they can be extended to further matches of o_q^{KL} based on received o^{pre-KL} matches. The number of instances in each partial match class per time unit equals the rates of the respective queries for which matches are maintained in the class. Thus, the resulting comparison costs of binary Kleene Closure evaluation are:

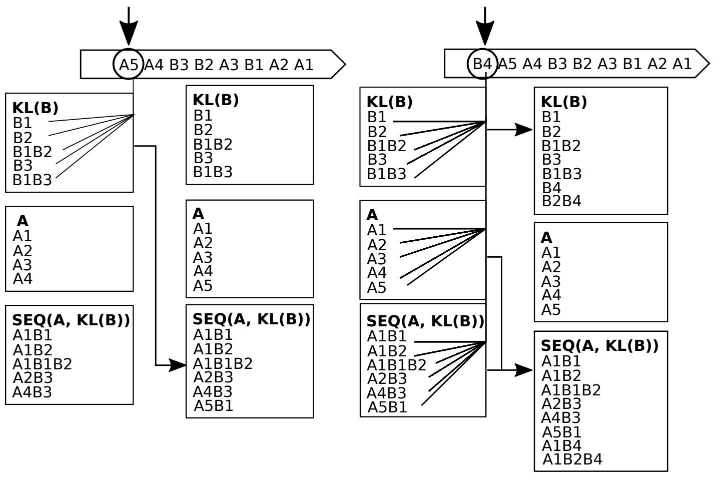
$$\begin{aligned}\kappa_{\hat{o^{KL}}} &= c \cdot (\frac{r(o_q^{pre-noKL})}{(|o_{pu}^{KL}| - 1) \cdot o_{part}^{pre-noKL} + 1} \cdot r(q_{KL}) + \\ c \cdot r(o_q^{pre-KL}) &\cdot (\frac{r(o_q^{pre-noKL})}{(|o_{pu}^{KL}| - 1) \cdot o_{part}^{pre-noKL} + 1} + r(q_{KL}) + r(o_q^{KL}))\end{aligned}$$

Example 7. *Fig. 7 shows both cases of the binary matching procedure for which matches of q are generated based on A and B events. To this end, besides the partial match class containing matches of q, which again can be extended to further matches of q, a second partial match class contains the received instances of A events, and a third class contains matches of the query KL(B). If an A event is received, it is compared against all partial matches of type KL(B) to form new matches of q which are sent to a potential next operator and kept in state for further matching. If a B event is received, it is compared against matches in all classes to create new matches of KL(B) or extend matches from class A or q to further matches of q.*

5.4 Correctness and Optimality

For this section, let $PG = (V_O, E_O)$ be a DecoPa plan for the evaluation of the query q on the processing units P. The maximal ingestion rate of each processing unit in P is given by ι and the maximal comparison rate that can be achieved for the evaluation of the predicates q_π of the query is given by κ. Moreover, let D be the distribution of occurrences of the event types referenced in q.

Correctness. To enable correct parallel query evaluation, our objective is to ensure, that the execution of the DecoPa plan PG on the processing units P yields all matches of q at the processing



A5 A4 B3 B2 A3 B1 A2 A1
KL(B) B1 B2 B1B2 B3 B1B3
A A1 A2 A3 A4
SEQ(A, KL(B)) A1B1 A1B2 A1B1B2 A2B3 A4B3
KL(B) B1 B2 B1B2 B3 B1B3
A A1 A2 A3 A4 A5
SEQ(A, KL(B)) A1B1 A1B2 A1B1B2 A2B3 A4B3 A5B1
B4 A5 A4 B3 B2 A3 B1 A2 A1
KL(B) B1 B2 B1B2 B3 B1B3
A A1 A2 A3 A4 A5
SEQ(A, KL(B)) A1B1 A1B2 A1B1B2 A2B3 A4B3 A5B1
KL(B) B1 B2 B1B2 B3 B1B3 B4 B2B4
A A1 A2 A3 A4 A5
SEQ(A, KL(B)) A1B1 A1B2 A1B1B2 A2B3 A4B3 A5B1 A1B4 A1B2B4

Fig. 7. Comparisons for binary Kleene Closure matching.

units of the operator $root(PG)$. To this end, the processing units of each operator $o \in V_O$ must jointly generate all matches of o_q.

Definition 5 (Structural Validity). *The DecoPa plan PG is* structural valid*, if the following properties are satisfied:*

1. *There exists a correct split $s(q) = (S_q, E_q)$ of the query q such that $i\colon V_O \longrightarrow S_q$ with $\forall o \in V_O, i(o) = o_q$ yields an isomorphism between the vertex sets V_O and S_q .*
2. *For all operators $o \in V_O$, there exists at most one predecessor in $o^{pre} \in pred(o)$ for which $o^{pred}_{part} = 1$.*
3. *Let o^{KL} denote an operator which matches Kleene Closure for ϵ_{KL} with a binary split, and $o^{\epsilon_{KL}} \in pre(o^{KL})$ denote o^{KL}'s predecessor for which the respective query comprises ϵ_{KL}, then $o^{\epsilon_{KL}}_{part} = 0$ must hold.*

The definition of a structural valid DecoPa plan implies correctness:

Theorem 1. *Evaluating a structural valid DecoPa plan for a query q and the processing units P, all matches of q are generated at a subset of P.*

Proof. (sketch.) Because of (1.), for each operator o, based on the matches of preceding operators, correct matches of o_q can be generated. Hence, $root(PG)$ generates correct matches of q. To enable complete evaluation for an operator o having two preceding operators o^i, o^j, each match of o^i_q must be combined with each match of o^j_q by at least one processing unit of o. This may not be the case, if a processing unit receives only partitions of the matches of both o^i_q and o^j_q. Though, this case is excluded by (2.). For correct Kleene matching all events of the respective Kleene type must be processed by the same processing unit, which follows from (3.). □

Throughput Validity. Given a scaling factor c which scales the occurrences D of the event types in q. The parallelization plan PG is *throughput valid* for the scaling factor, when all processing units involved in the execution of PG yield sustainable throughput.

Definition 6 (Throughput Validity). *PG is* throughput valid *for $c \in \mathcal{R}$, if*

$$\kappa \leq \max\{\hat{\kappa_o} \mid o \in V_0\} \wedge \iota \leq \max\{\hat{\iota_o} \mid o \in V_0\}.$$

Optimality. A parallelization plan PG is *optimal*, if it maximizes the sustainable throughput.

Definition 7 (Optimality). *Let the evaluation plan PG for the evaluation of q on P processing instances be structural valid and throughput valid for c. PG is* optimal, *if there exists no other structural valid parallelization plan PG' for q, P such that PG' is throughput valid for a scaling factor c' and $c' > c$.*

To construct an optimal DecoPa plan, a split and respective resources assignment must be determined, such that the maximal comparison and input costs of the resulting operators are minimized (Def. 7). Even for a restricted version, which considers only *ld* splits for the plan construction, the problem of constructing optimal DecoPa plans is NP-hard, which can be shown by a reduction from optimal join-ordering for left-deep trees which is known to be NP-hard [10].

Theorem 2. *Optimal DecoPa plan construction is NP-hard.*

Proof. (sketch). For a join-ordering instance comprising the relations $R_1, \ldots, Rn$ and the pairwise selectivities Σ of the join predicates, we can construct a *ld*-DecoPa plan using $|R_1|, \ldots, |R_n|$ as event rates together with Σ to instantiate the cost functions. We choose $\iota = 0$ and assume enough resources to balance out the comparisons of the operators, such that the comparison costs of all operators are equivalent with the consequence, that to maximize c, the sum of comparison costs of its operators needs to be minimized. The underlying split of the optimal DecoPa plan optimizes the join-ordering for the given problem instance. □

5.5 Generalization of DecoPa Plans

Multi-Query Setting. The definition of DecoPa plans extends naturally to workloads of queries. Let $PG_1, \ldots, PG_n$ be DecoPa plans for the queries of the workload $Q = \{q_1, \ldots, q_n\}$. Unifying the sets of operators and edges of the DecoPa plan, we obtain a DecoPa plan PG_Q for the workload having a set of root operators for the queries in Q. If a multi-query DecoPa plan PG_Q has multiple operators referencing the same sub-query s_q, processing units can be shared by keeping only one operator for s_q. In the next section, we discuss strategies for promoting the sharing of sub-queries and processing units during the construction of multi-query DecoPa plans.

Adaptivity. The effectiveness of a DecoPa plan for a query relies on the rates of the event types of the query and the predicate selectivities. A change thereof over time is identified by a processing unit, if the number of matches of the query evaluated deviates from the predicted rates. In that case, a new DecoPa plan is constructed and deployed, while we leave the adaptation of a DecoPa plan through fine-granular substitution of sub-queries to future work.

6 CONSTRUCTION

As the optimal DecoPa construction is intractable, this section is devoted to the efficient construction thereof. We discuss preliminary considerations on the partitioning, resource assignment, and sub-query selection (§6.1), before turning to the decomposition of a query into a split graph and the subsequent DecoPa plan instantiation (§6.2). Then, we elucidate the extension to a multi-query scenario (§6.3) and state the complexity of our construction algorithm (§6.4).

6.1 Preliminary Considerations.

Resource Assignment. The number of available processing units influences the choice of splits that can be used for the construction of a DecoPa Plan, as shown in Fig. 2: If only 2 processing units were available instead of 4, the right plan could not be used for the evaluation of q. Let $s(q) = (S_q, E_q)$ be a split for the query q, P a set of processing with ι and κ, and c a scaling factor. Moreover, let $S_{q_c} \subset S_q$ denote the sub-queries defined over more than one primitive event type. Given that $|P| > |S_{q_c}|$, we assign the resources to each sub-query respecting ι and κ: Let $s_q \in S_{q_c}$ be a (non-Kleene) sub-query with a binary split having pre_1, pre_2 as preceding sub-queries in $s(q)$, and let $r(pre_1) \geq r(pre_2)$. The resources allocated to s_q are given by: $pu(s_q) = \max\{\frac{pre_1 \cdot pre_2 \cdot 2}{\kappa}, \frac{1}{(\iota - pre_2)} \cdot pre_1\}$. For Kleene matching, the numerator of the left part reflects the respective comparison costs as discussed in §5.3. If there are not enough processing units, i.e., $|P| < \sum_{s_q \in S_{qc}} pu(s_q)$, no throughput valid plan can be constructed for c based on the given split. Remaining resources $|P| - \sum_{s_q \in S_{qc}} pu(s_q)$ are distributed among the sub-queries proportionally, to approximate equivalent comparison costs for each operator.

Partitioning Scheme. If an operator o has more than one processing unit assigned, the evaluation of o_q is jointly done by the processing units o_{pu}. As we only consider the construction of structural valid DecoPa plans, at most one of the preceding operators $pre_1, pre_2 \in pred(o)$ can have $pre_{i_{part}} = 1$ (see Def. 5). To minimize the input costs $\hat{\iota_o}$ of each operator, we construct DecoPa plans, such that the operator pre_1, for which $r(pre_{1_q}) > r(pre_{2_q})$, partitions its matches. The comparison costs $\hat{\kappa_o}$ are unaffected by this choice.

Rate-Reducing Sub-Queries. As the comparison and input costs of operators in a DecoPa plan are defined based on the match rates of the sub-queries in the underlying query split, we consider only *rate-reducing* sub-queries for the construction of query splits: That is, we restrict the search space by considering only sub-queries s_q, which have an output rate lower than the maximal rate of the comprised primitive event types, i.e., $r(s_q) < \max_{\epsilon \in types(q)} r(\epsilon)$. This way, the negative impact of high rate event types on the comparison and input costs of a DecoPa plan is contained.

6.2 DecoPa Plan Instantiation

As shown in Alg. 1, we employ dynamic programming to construct DecoPa plans. The function `Plan`(s, PU) instantiates a DecoPa Plan for a given split s and processing units PU, assigning the processing units and partitioning as explained in the previous section. To compare plans, we use the cost function `costs`(P) to assign to a plan P its sum of comparisons. In a first step, all rate-reducing sub-queries SQ are enumerated for q (line 1). As the rate-reducing sub-queries may not suffice to construct a complete binary split for the query, a set of ld splits is sampled for q and each sub-query in SQ (line 3). For each sub-query, we determine the ld split inducing the least costs when instantiated in a (ld-)DecoPa plan, assigning one processing unit to each sub-query (line 2). Based on the best ld split and the respective costs, two maps, one for keeping the current best split (*Splits*), one for keeping the minimal costs (*PCosts*), are initialized for each sub-query and q (line 5-line 6).

In the main loop, we iterate over each sub-query $sq \in SQ$ sorted in a topological order (line 7): First, we compute for sq a number of processing units, which we use to instantiate candidate DecoPa plans (line 8). That is, we assign to sq processing units proportional to the costs of the best (ld-)DecoPa plan with respect to the costs of the (ld-)DecoPa of all other sub-queries. Then, *PCosts* of sq are updated with the costs of the best ld-DecoPa plan instantiated with this number of processing units (line 9). The respective number of processing units does not necessarily equal the final number of processing units assigned to sq when used in a Plan and only serves as heuristic for the cost function during the split enumeration. Next, the set of predecessor candidates *PredC* given by the rate-reducing sub-queries of sq and for which the determined costs are less than the currently estimated costs of sq (line 10) is computed. Due to the ordered processing, final splits and respective costs for each potential predecessor were already constructed in an earlier iteration. Then, for each tuple in *PredC* which yields a potential set of predecessors of sq in a structural valid split, i.e, covers the event types of sq, a split is constructed by combining the splits of the predecessors in a split graph having sq as root (line 12 - line 13). The costs of a plan instantiating this potential split *SplitC* with the predetermined processing units are computed and the split for

Algorithm 1: DecoPa Construction.

```
input  : query $q$; processing units $totalPU$
output : DecoPa plan
/* Generate set of rate-reducing sub-queries. */
1  $SQ \leftarrow$ getRateReducingSQs($q$)
2  for $sq \in SQ \cup q$ do                                        // For each sub-query and $q$
3      $ldSplits \leftarrow$ sampleLdSplits($sq$) ;                   // Sample ld splits of $sq$
4      for $ld \in ldSplits$ do                                    // For each ld split
5          if costs(Plan($ld$, $|types(sq) - 1|$)) $< PCosts[sq]$ then
6              $PCosts[sq] \leftarrow$ costs(Plan($ld$, $|types(sq) - 1|$)) $Splits[sq] \leftarrow ld$
7  for $sq \in$ topoSort($SQ \cup q$) do                             // Iterate over sorted $SQ$
       /* Assign processing units proportionally to sq */
8      $PU[sq] \leftarrow (PCosts[sq] / \sum_{i \in SQ} PCosts[i]) * |totalPU|$
       /* Adjust plan costs with processing units */
9      $PU[sq] \leftarrow$ costs(Plan($ld$, $PU[sq]$))
       /* Determine predecessor candidates */
10     $PredC \leftarrow \{x | x \in SQ \wedge types(x) \subset types(sq) \wedge PCosts[x] < PCosts[sq]\}$
       /* Loop over candidate predecessors */
11     for $(s1, s2) \in \{(s1, s2) \mid s1, s2 \in PredC \wedge$
12     $s1 \neq s2 \wedge types(s1) \cup types(s2) = types(sq)\}$ do
           /* Split graphs of $s1$, $s2$ */
13         $G_{s1} = (V_1, E_1), G_{s2} = (V_2, E_2) \leftarrow Splits[s1], Splits[s2]$
           /* Construct candidate split of $G_{s1}$, $G_{s2}$ */
14         $SplitC = ((V_1 \cup V_2 \cup \{sq\}), (E_1 \cup E_s \cup \{(sq, s1), (sq, s2)\})$
           /* Update, if $SplitC$ improves costs */
15         if costs(Plan($SplitC$, $PU[sq]$) $< PlanCosts[sq]$ then
16             $PCosts[sq] \leftarrow$ costs(Plan($SplitC$, $PU[sq]$) $Splits[sq] \leftarrow SplitC$
17     return Plan($Splits[q]$, $totalPU$);
```

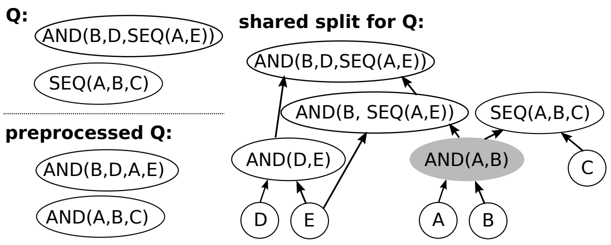


Fig. 8. Shared split construction for the query workload Q.

the plan introducing minimal costs is kept (line 15 - line 16). In the last iteration, the query q is processed and a respective split is constructed.

The total number of processing units are assigned to the sub-queries in the split to finalize the DecoPa plan for the query (line 17).

6.3 Multi-query Construction

To efficiently parallelize the evaluation of multi-query workloads, we enhanced our construction to foster sharing of operators between queries and assign processing units proportionally with respect to all queries in Q. Fig. 8 shows phases of the shared split construction of the workload Q.

If the queries differ in their time windows and predicates, the query workload can be 'normalized' by choosing the maximal time window for the evaluation and, for each tuple of event types, the disjunction of predicates thereof within all query predicates of the workloads queries. Consequently, an operator evaluating the query q of a normalized workload additionally needs to check q's time window and predicates on each produced match of q. As a further preprocessing step, each query q is transformed to a query q' for which the pattern trees' structure comprises a root pattern function f with $f.sem = AND$ with the children $\varphi(f)$ being $q's$ primitive pattern functions. The resulting queries have no order constraints, which allows to leverage sharing opportunities beyond syntactic equivalence. Starting with the query for which a DecoPa plan having an *ld* split has maximal comparison costs, Alg. 1 is run twice for each query. In the first round, the considered sub-queries are restricted to those already used in a split of a query processed in an earlier iteration. For the second round, all sub-queries are considered. The resulting split inducing lower costs is chosen for each query. After a split has been constructed for each query, all sub-queries only used for one query are transformed back with respect to their original query to avoid higher output rates induced by the *AND* pattern function. The final assignment of processing units for the DecoPa plan of the workload is done after all query splits have been constructed, such that processing units are shared for sub-query evaluation. Our multi-query construction is incremental, so that reusing existing operators in an updated DecoPa plan for a modified workload is possible if the execution environment allows for adaptive query scheduling.

6.4 Complexity of Construction

To enumerate the set of rate-reducing sub-queries of a query q, each subset of $types(q)$ needs to be considered which yields a time complexity of $O(2^{|types(q)|})$. For the construction of the DecoPa plan, k *ld* splits are sampled for each rate-reducing sub-query, introducing additional time complexity of $O(k \cdot 2^{|types(q)|})$. In the main loop, to construct the split, for each sub-query sq, tuples of sub-queries of s are enumerated, yielding a time complexity of $O(2^{|types(q)|} \cdot (2^{|types(q)|})^2)$.

7 EVALUATION

We evaluated DecoPa plans for parallel CEP query evaluation in several experiments. Below, we review our setup (§7.1), before turning to a state-of-the-art comparison (§7.2). Then, we present our results for real-world data (§7.3), and a sensitivity analysis based on synthetic data (§7.4).

7.1 Evaluation Setup

Data Sets and Queries. We evaluated DecoPa plan based query evaluation on two real-world data sets: the Google Cluster traces [29], and the Citi Bike data set [9]. Moreover, we used synthetic data in a sensitivity analysis to achieve a controlled setup for investigating the influence of different workload characteristics.

The Google Cluster data set has 9 event types, referring to state transitions in the task life-cycle of jobs. The Citi Bike data set contains events denoting bike rentals from which we derived 9 event types based on the age, customer type, and rental length. For each data set, we evaluated three queries of which two comprise different nestings of the *AND* and *SEQ* pattern function, and one comprises Kleene Closure. For the Google Cluster trace, we used a sub-trace of the events generated in 48 hours comprising 7 million events; for Citi Bike, the queries were evaluated on a trace covering bike rentals within a month, comprising 1 million events. The predicate used in the Google Cluster queries requires decreasing values for the requested CPU- and memory capacity, and decreasing priority for consecutive events in a match. For the the Citi Bike queries, the predicate is satisfied if the distance between the start locations of each tuple of rides is greater than 6 kilometers. That is, to be considered as a match of a query, a set of rides needs be geographically dispersed. The

Google Cluster queries are evaluated over a time window of 10 hours; for Citi Bike, we used a time window of 24 hours. The predicates yielded a selectivity of 0.07% (Google Cluster) and 0.05% (Citi Bike). Both data sets showed skewed event rate distributions, with rate differences of up to 5 orders of magnitude.

For the synthetic data, we generated different event rate distributions which were drawn from a Zipfian process with a skew parameter set to 1.2 resulting in fairly skewed event rates. Based on the event rates, we instantiated a Poisson process to generate event traces. We constructed queries with varying lengths and semantics as well as query workloads of different sizes. To simulate query predicates, we drew uniformly selectivities from the range $1 - 10\%$. If not stated otherwise, the time window used was 1 minute.

Baselines. None of existing approaches for parallel CEP considers decomposition. Thus, for all baseline strategies, we used an *ld* split with the ordering directly derived from the respective query. As a centralized baseline, all sub-queries of the split are processed by a single processing unit. In a state-based baseline (*Stateparallel*), each sub-query is evaluated by a single processing unit. As a baseline for hybrid approaches [30, 31], we assigned to each sub-query processing units relative to the expected load in terms of comparison costs and chose to partition the maximal input of each operator among its processing units (*Hybrid*). RIP [5] was proposed for query evaluation with a contiguous event selection strategy and a given maximal match-length. To adapt RIP to our setting (*RIP*-inspired), we assigned batches of 2 time windows to processing units in a round-robin manner with an overlap of one window for any two consecutive batches.

While we used FlinkCEP as an evaluation engine, we note that FlinkCEP does not (auto-)parallelize CEP patterns but can be applied on a key-partitioned stream. To support arbitrary predicates, key-based parallelization is insufficient, which is why we did not include native FlinkCEP in our comparison.

Metrics. If not stated otherwise, we compared the DecoPa based query evaluation against the baseline approaches in terms of the resulting throughput gain over a centralized query evaluation. To this end, we determined the maximal scaling factor $c_{centralized}$ for a given problem instance that could be achieved for centralized evaluation and the scaling factor c_{other} for a respective other approach and report on the ratio $c_{other}/c_{centralized}$. Moreover, we measured the detection latency resulting from the parallel evaluation.

Implementation. We implemented our construction algorithm and cost models in Python and deployed the resulting DecoPa plans on processing units running FlinkCEP [7][4]. The shuffling process that can be instantiated based on a DecoPa plan was not part of our setup, but it was realized by a preprocessing step that generated event traces for each processing unit using a DecoPa plan.

To enhance flexibility, event sources were decoupled from the FlinkCEP instances, and events were exchanged between sources, processing units, and among processing units using TCP sockets. We investigated the DecoPa based query evaluation in a single-node server-based setup on a NUMA node with 4 Intel Xeon E7-4880 (60 cores, 1TB RAM) and in a cluster-based setup comprising 20 Raspberry Pi 4B nodes.

Before generating DecoPa plans specific to each scenario, we measured the ingestion and comparison rates across different workloads and execution environments. We measured the time interval between two predicate checks to determine the comparison rate for real-world data sets. For synthetic data, we introduced delays to simulate different values for κ. To estimate the ingestion rate, we monitored the delay between the source function and the CEP operator for a growing number of incoming events.

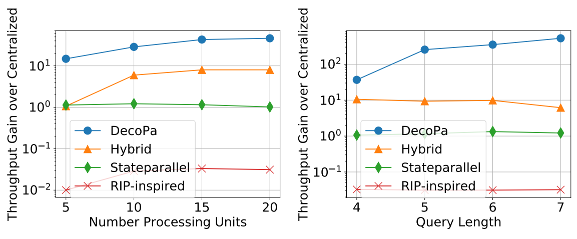
[4]The codes is publicly available at https://github.com/samieze/DecoPa.

Table 3. Throughput gain for real-world data sets.

		Citi Bike			Google Cluster		
		Q1	Q2	Q3	Q1	Q2	Q3
5	DecoPa	52.4×	212.8×	758.5×	578.0×	13.1×	3.1×
	Hybrid	1.0×	2.9×	155.2×	2.1×	1.2×	3.1×
10	DecoPa	204.9×	436.3×	3714.8×	1989×	20.1×	8.2×
	Hybrid	3.1×	3.7×	155.2×	2.2×	1.2×	5.1×
15	DecoPa	370.3×	899.5×	6508.7×	3411×	33.0×	13.3×
	Hybrid	5.3×	6.4×	311.5×	2.2×	1.2×	8.0×
20	DecoPa	428.1×	1442.2×	8585.7×	4209×	53.4×	18.8×
	Hybrid	5.3×	10.7×	311.5×	2.2×	1.2×	10.2×

7.2 State-of-the-Art Comparison

Fig. 9 shows the throughput gain over centralized for DecoPa and all other baseline strategies. In Fig. 17a, a query of length 4 is evaluated with a varying number of processing units available. DecoPa achieves the highest throughput gain, followed by Hybrid, with both approaches scaling in the number of processing units. This can be also observed for our RIP inspired baseline, although the throughput achieved by the approach is less than what can be achieved by the centralized baseline as more events need to be processed per unit due to the batch-based parallelism. As expected, the throughput-gain of state-based parallelism is independent of the number of processing units. The highest throughput gain for DecoPa is observable when raising the number of processing units from 5 – 15, which allows to use more intricate splits for the construction.



(a) (b)

Fig. 9. Comparison against state-of-the-art approaches.

Fig. 17b illustrates the throughput gain for queries of varying length with the number of processing units set to 15. DecoPa achieves a significant throughput gain of more than 2 orders of magnitude for queries of length of 7, which is due to longer queries providing more possibilities for decomposition. The other approaches are equally affected as the centralized evaluation by a varying query length with a slight downtrend visible for Hybrid. With all baselines using the same *ld*-split, there is often one sub-query posing a bottleneck, which equally constrains all approaches. In the remainder, we compare DecoPa only against Hybrid as it achieved the best results among all baseline strategies.

7.3 Experiments with Real-World Data Sets

The experiments were conducted in the Raspberry Pi cluster, deploying one processing unit per Raspberry Pi. Here, we determined a maximal ingestion rate ι of 2500 events per minute, and measured the maximal comparison rate κ for each data set based on the respective predicates. Listing 1 shows the three queries used for the Citi Bike data set. With the predicates for Citi Bike involving more complex computations, we used $\kappa = 80000$ for the Citi Bike queries and $\kappa = 500000$ for the Google Cluster queries.

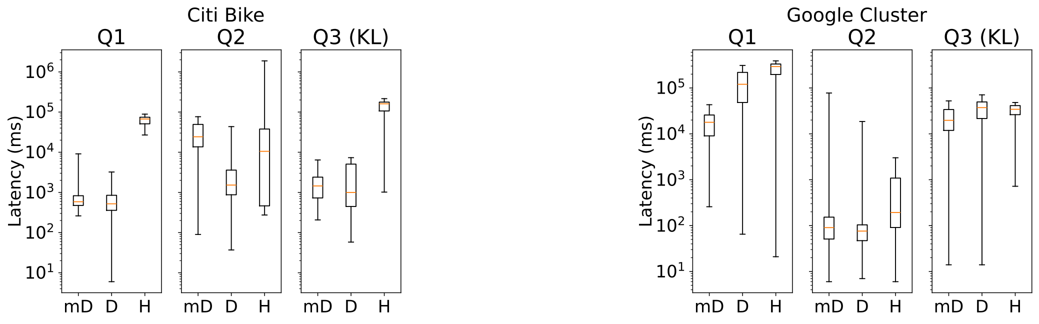


Fig. 10. Latency comparison for real-world data sets.

Table 3 shows the throughput gain for DecoPa and Hybrid over centralized for the three queries of each data set with varying numbers of processing units. For both data sets, DecoPa based evaluation achieves a throughput gain of up to 3 orders of magnitude compared to centralized and Hybrid.

In general, it can be observed that DecoPa scales well with the number of processing units for all evaluated queries, with $Q1$ of the Citi Bike data set even quadrupling its throughput gain, when processing units are doubled. Hybrid did not scale for the queries $Q1$ and $Q2$ of the Google Cluster data set when adding resources due to the very high-rate event types in the Google Cluster data. Combining two of the latter results in high comparison costs, which can not be reduced by assigning only $5 - 10$ more resources. $Q3$ of both data sets contains Kleene Closure. While the sensitivity analysis shows that due to using unary splits, DecoPa plans can significantly optimize throughput for queries with Kleene, this result is not as pronounced for the real-world data sets as we could define Kleene Closure only on medium- or low-rate event types as otherwise, no practically usable scaling factor could be achieved for the evaluation.

We compared DecoPa and Hybrid in terms of latency (Fig. 10), by evaluating (1) the throughput-maximizing DecoPa plan on the event rates scaled with the respective scaling factor (mD), (2) a DecoPa plan on the event rates scaled with the maximal scaling factor obtained from Hybrid (D), and (3) the Hybrid plan for the rates scaled with its respective maximal scaling factor (H).

For all queries, except $Q2$ of Citi Bike, D *and* mD yield lower latencies than H, which is a remarkable result as, e.g., for $Q1$ of Citi Bike, almost 100× more throughput is achieved while minimizing latency by 2 orders of magnitude. As Kleene is defined over fairly low rates, even in the DecoPa plans, binary splits were used, resulting in equivalent latencies for $Q3$ of Google Cluster for all experiments. The Kleene query $Q3$ of Citi Bike minimized latency by 2 orders of magnitude.

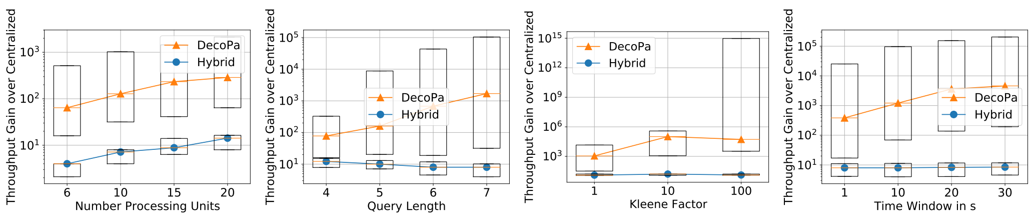
7.4 Sensitivity Analysis

Our analysis focuses on the influence of different query characteristics and processing contexts on the throughput gain and latency. The experiments were conducted in the server-based setup, where processing units were instantiated by the server cores, which yielded a maximal ingestion rate of $\iota = 6500$. We simulated predicate evaluation with a resulting $\kappa = 60000$, if not stated otherwise. The results represent the median values obtained from a set of 50 experiments with the boxes in Fig. 11 depicting the 25th and 75th percentiles.

Fig. 12 provides an overview of the throughput gain in comparison to Hybrid across different $\kappa = 60k$ and $\kappa = 600k$, with a lower value (more complex predicate evaluation) yielding more optimization potential. For each κ, plans were constructed for queries in which the event types referenced in the query were ordered increasingly based on their rates, which denotes the worst-case

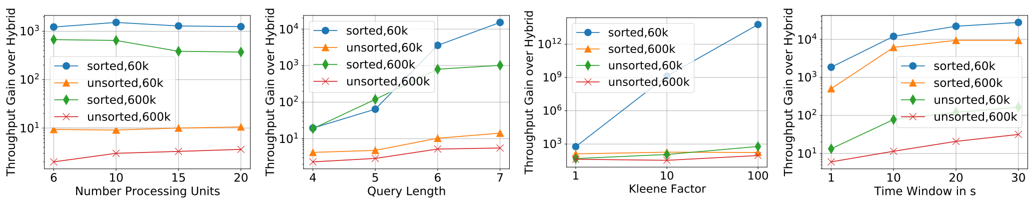
```
Query 1 - Citi Bike:
PATTERN AND(ShortY sy,VLongO vl,ShortO so,LongO lo,LongC lc)
WHERE ∀(i, j) ∈ {sy, vl, so, lc}^2, i ≠ j ∧ dist(i.Loc, j.Loc) ≥ 6km
WITHIN 24h
Query 2 - Citi Bike:
PATTERN SEQ(ShortY sy,LongC lc, AND(LongY ly,VLongY vly))
WHERE ∀(i, j) ∈ {sy, lc, ly, vly}^2, i ≠ j ∧ dist(i.Loc, j.Loc) ≥ 6km
WITHIN 24h
Query 3 - Citi Bike:
PATTERN AND(LongY ly,KL(ShortY sy),ShortO so,LongC lc)
WHERE ∀(i, j) ∈ {ly, sy, so, lc}^2, i ≠ j ∧ dist(i.Loc, j.Loc) ≥ 6km
WITHIN 24h

Query 1 - Google Cluster:
PATTERN AND(Submit s,SEQ(Evict e1,Enable e2),Finish f)
WHERE ∀(i, j) ∈ {s, e1, e2, f}^2, i.mem_usage ≥ j.mem_usage ∧ i.cpu_usage ≥ j.cpu_usage
WITHIN 10min
Query 2 - Google Cluster:
PATTERN AND(Enable e1,Enable e2,Lost l,Update u)
WHERE ∀(i, j) ∈ {e1, e2, l, u}^2, i.mem_usage ≥ j.mem_usage ∧ i.cpu_usage ≥ j.cpu_usage
WITHIN 10min
Query 3 - Google Cluster:
PATTERN AND(KL(Schedule s),Queue q,Lost l)
WHERE ∀(i, j) ∈ {s, l, q}^2, i.mem_usage ≥ j.mem_usage ∧ i.cpu_usage ≥ j.cpu_usage
WITHIN 10min
```

Listing 1. Queries used for real-world data sets.



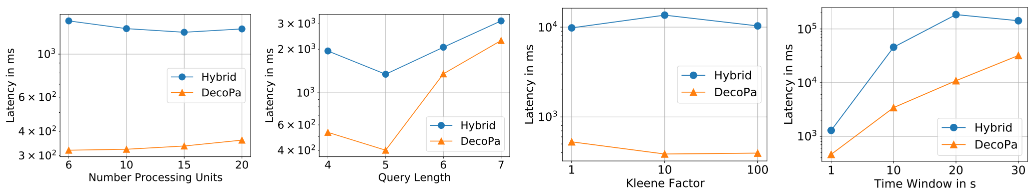
(a) DoP (b) Query Length (c) Kleene Closure (d) Time Window

Fig. 11. Throughput gain over centralized.



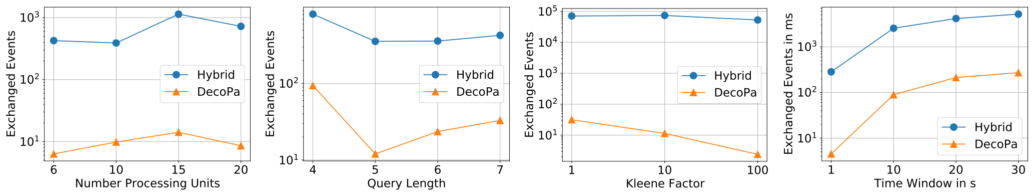
(a) DoP (b) Query Length (c) Kleene Closure (d) Time Window

Fig. 12. Throughput gain over Hybrid.



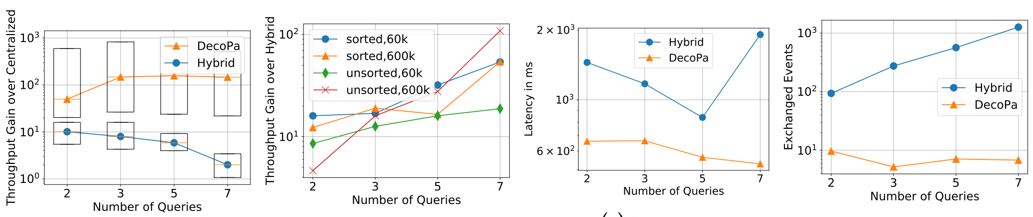
(a) DoP (b) Query Length (c) Kleene Closure (d) Time Window

Fig. 13. Comparison of detection latency of DecoPa and Hybrid for maximal scaling factor of Hybrid.



(a) DoP (b) Query Length (c) Kleene Closure (d) Time Window

Fig. 14. Comparison of exchanged events of DecoPa and Hybrid for maximal scaling factor of Hybrid.



(a) Throughput over central. (b) Throughput over Hybrid (c) Latency (d) Network Costs

Fig. 15. Results of Multi-Query Experiments.

for Hybrid. The combination of lower κ values and ordered events produced the most substantial throughput optimization. As discussed above, DecoPa scales well with increasing resources with the most optimization potential when raising the number of processing units from $5-15$.

In Fig. 13, we compare DecoPa and Hybrid in terms of induced latency. To this end, we constructed DecoPa plans for the maximal scaling factor that could be determined for Hybrid. The results do not include the time needed to construct a DecoPa plan (which is separately shown in Fig. 16). The most significant improvements in detection latency are visible for the Kleene Closure experiments with differences of more than one order of magnitude, which are caused by the evaluation based on unary splits requiring fewer comparisons.

Query Length. Increasing query lengths yield the highest optimization potential for decomposition, with the number of possible sub-queries available growing exponentially in the query length. As such, a throughput gain of 5 and 4 orders of magnitude could be achieved against centralized and Hybrid, respectively.

Time Window. We varied the time windows for the query evaluation from $1-30$ minutes. Longer time windows increase the number of partial matches and, thus, influence the resulting comparison costs of each operator. As rate-reducing sub-queries constrain the number of partial matches maintained per operator, a throughput gain of up to 5 orders of magnitude was achieved in this experiment. For this parameter, naturally, the highest latencies can be observed.

Kleene Closure. To investigate the impact of Kleene Closure, we varied the rate of the event type upon which Kleene Closure is defined by setting it to $1\times$, $10\times$, and $100\times$ the average rate of the remaining types of the query. While DecoPa plans mostly employed unary splits, the Hybrid approach, underlying an *ld* split, naturally employs binary Kleene splits. They entail more comparisons due to the three classes of partial matches to maintain, which was confirmed in our results, reaching up to 12 orders of magnitude throughput gain and 1 order of magnitude lower latencies compared to Hybrid.

Multi-Query. In Fig. 15, we study multi-query scenarios by constructing DecoPa plans for workloads of up to 7 queries. Sharing operators between queries allows us to make more use of the available processing units, which is reflected in the results for throughput gain, which scales proportionally compared to Hybrid, where the possible degree of parallelization per sub-query decreases with a growing number of queries with no sharing being exploited.

Network Costs. In Fig. 14, the network costs, i.e., the number of events exchanged between all processing units per time window, are shown. Although more events are processed (i.e., as shown by the overall throughput gain of DecoPa), the network costs are lower by orders of magnitude as only the (fewer) matches of rate-reducing sub-queries are exchanged between the processing units.

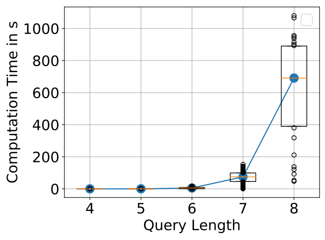


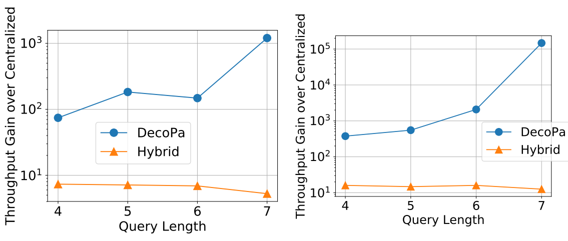
Fig. 16. Plan construction time.

Optimization Time. Fig. 16 shows the exponential correlation between DecoPa plan construction time and query length. Yet, the construction time increases in the number of rate-reducing sub-queries considered for the split construction. Although longer queries may offer more rate-reducing sub-queries, selectivity is the key factor influencing this trend.

Results on Bounds. We constructed DecoPa plans for queries with varying length for an ingestion-bound scenario ($\kappa = 500k, \iota = 2k$) and comparison-bound scenario ($\kappa = 60k, \iota = 15k$). We chose the configurations as $\kappa = 60k$ captures, for a time window of 1 minute, that a single comparison takes $1ms$, which we observed for simple predicate checks in our experiments. Moreover, even for the server-based setup, the maximal number of events that could be ingested from a socket-based source to FlinkCEP was around $10k$ per minute. Fig. 17 shows that for both scenarios, notable improvements materialize.

8 CONCLUSIONS

In this paper, we proposed DecoPa plans for the parallel evaluation of CEP workloads. Compared to existing approaches, DecoPa plans consider event rates and the processing context to construct throughput-maximizing query splits, tailored for parallel evaluation. We formalized DecoPa plans and discussed properties thereof. With optimal DecoPa plan construction being NP-hard, we propose a dynamic programming based construction as well as strategies to cater for multi-query scenarios. Our experimental evaluation showed the benefits of DecoPa plans compared to state-of-the art approaches. Especially for computationally demanding query characteristics, such as long time windows and Kleene Closure, we achieved a throughput gain by up to multiple orders of magnitude over baseline strategies.



(a) Ingestion-bound (b) Comparison-bound

Fig. 17. Optimization potential for varying query length in comparison-bound and ingestion-bound scenarios.

REFERENCES

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 147–160. https://doi.org/10.1145/1376616.1376634

[2] Samira Akili, Steven Purtzel, and Matthias Weidlich. 2023. INEv: In-Network Evaluation for Event Stream Processing. *Proc. ACM Manag. Data* 1, 1 (2023), 101:1–101:26. https://doi.org/10.1145/3588955

[3] Samira Akili and Matthias Weidlich. 2021. MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 10–22. https://doi.org/10.1145/3448016.3457318

[4] Alexander Artikis, Matthias Weidlich, François Schnitzler, Ioannis Boutsis, Thomas Liebig, Nico Piatkowski, Christian Bockermann, Katharina Morik, Vana Kalogeraki, Jakub Marecek, Avigdor Gal, Shie Mannor, Dimitrios Gunopulos, and Dermot Kinane. 2014. Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 712–723. https://doi.org/10.5441/002/edbt.2014.77

[5] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. RIP: run-based intra-query parallelism for scalable complex event processing. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, Sharma Chakravarthy, Susan Darling Urban, Peter R. Pietzuch, and Elke A. Rundensteiner (Eds.). ACM, 3–14. https://doi.org/10.1145/2488222.2488257

[6] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. 2009. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, Aniruddha S. Gokhale and Douglas C. Schmidt (Eds.). ACM. https://doi.org/10.1145/1619258.1619263

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[8] Sharma Chakravarthy and D. Mishra. 1994. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowl. Eng.* 14, 1 (1994), 1–26. https://doi.org/10.1016/0169-023X(94)90006-X

[9] citi Bike. 2023. http://www.citibikenyc.com/system-data..

[10] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 893)*, Georg Gottlob and Moshe Y. Vardi (Eds.). Springer, 54–67. https://doi.org/10.1007/3-540-58907-4_6

[11] Luping Ding, Karen Works, and Elke A. Rundensteiner. 2011. Semantic stream query optimization exploiting dynamic metadata. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 111–122. https://doi.org/10.1109/ICDE.2011.5767840

[12] EsperTech. 2023. https://www.espertech.com/esper/..

[13] 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). https://doi.org/10.1016/j.is.2019.101442

[14] 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. https://doi.org/10.1007/s00778-019-00557-w

[15] Martin Hirzel. 2012. Partition and compose: parallel complex event processing. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, François Bry, Adrian Paschke, Patrick Th. Eugster, Christof Fetzer, and Andreas Behrend (Eds.). ACM, 191–200. https://doi.org/10.1145/2335484.2335506

[16] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[17] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 589–606. https://doi.org/10.1145/3299869.3319869

[18] Ilya Kolchinsky, Assaf Schuster, and Danny Keren. 2016. Efficient Detection of Complex Event Patterns Using Lazy Chain Automata. *CoRR* abs/1612.05110 (2016). arXiv:1612.05110 http://arxiv.org/abs/1612.05110

[19] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy evaluation methods for detecting complex events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 34–45. https://doi.org/10.1145/2675743.2771832

[20] Tiziano De Matteis and Gabriele Mencagli. 2017. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *Int. J. Parallel Program.* 45, 2 (2017), 382–401. https://doi.org/10.1007/s10766-016-0413-x

[21] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 161–173. https://doi.org/10.1145/3135974.3135983

[22] 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 '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1452–1464. https://doi.org/10.1145/3448016.3452785

[23] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 495–510. https://doi.org/10.1145/2882903.2882947

[24] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Comput. Surv.* 52, 2 (2019), 36:1–36:37. https://doi.org/10.1145/3303849

[25] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, Aniruddha S. Gokhale and Douglas C. Schmidt (Eds.). ACM. https://doi.org/10.1145/1619258.1619264

[26] Kia Teymourian, Malte Rohde, and Adrian Paschke. 2012. Knowledge-based processing of complex stock market events. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari (Eds.). ACM, 594–597. https://doi.org/10.1145/2247596.2247674

[27] Di Wang, Elke A. Rundensteiner, Richard T. Ellison, and Han Wang. 2010. Active Complex Event Processing: Applications in Real-Time Health Care. *Proc. VLDB Endow.* 3, 2 (2010), 1545–1548. https://doi.org/10.14778/1920841.1921034

[28] Matthias Weidlich, Holger Ziekow, Avigdor Gal, Jan Mendling, and Mathias Weske. 2014. Optimizing Event Pattern Matching Using Business Process Models. *IEEE Trans. Knowl. Data Eng.* 26, 11 (2014), 2759–2773. https://doi.org/10.1109/TKDE.2014.2302306

[29] John Wilkes. 2020. Yet more Google compute cluster trace data. Google research blog. Posted at https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html..

[30] Fuyuan Xiao, Cheng Zhan, Hong Lai, Li Tao, and Zhiguo Qu. 2017. New parallel processing strategies in complex event processing systems with data streams. *Int. J. Distributed Sens. Networks* 13, 8 (2017). https://doi.org/10.1177/1550147717728626

[31] Maor Yankovitch, Ilya Kolchinsky, and Assaf Schuster. 2022. HYPERSONIC: A Hybrid Parallelization Approach for Scalable Complex Event Processing. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1093–1107. https://doi.org/10.1145/3514221.3517829

[32] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 217–228. https://doi.org/10.1145/2588555.2593671

Received October 2023; revised January 2024; accepted February 2024