Real-Time Multi-Pattern Detection over Event Streams

Ilya Kolchinsky
Technion, Israel Institute of Technology
ikolchin@cs.technion.ac.il

Assaf Schuster
Technion, Israel Institute of Technology
assaf@cs.technion.ac.il

ABSTRACT

Rapid advances in data-driven applications over recent years have intensified the need for efficient mechanisms capable of monitoring and detecting arbitrarily complex patterns in massive data streams. This task is usually performed by complex event processing (CEP) systems. CEP engines are required to process hundreds or even thousands of user-defined patterns in parallel under tight real-time constraints. To enhance the performance of this crucial operation, multiple techniques have been developed, utilizing well-known optimization approaches such as pattern rewriting and sharing common subexpressions. However, the scalability of these methods is limited by the high computation overhead, and the quality of the produced plans is compromised by ignoring significant parts of the solution space.

In this paper, we present a novel framework for real-time multi-pattern complex event processing. Our approach is based on formulating the above task as a global optimization problem and applying a combination of sharing and pattern reordering techniques to construct an optimal plan satisfying the problem constraints. To the best of our knowledge, no such fusion was previously attempted in the field of CEP optimization. To locate the best possible evaluation plan in the resulting hyperexponential solution space, we design efficient local search algorithms that utilize the unique problem structure. An extensive theoretical and empirical analysis of our system demonstrates its superiority over state-of-the-art solutions.

CCS CONCEPTS

• **Information systems → Stream management**; *Query optimization*;

KEYWORDS

Complex Event Processing, Multi-Query Optimization

ACM Reference Format:
Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. In *Proceedings of ACM SIGMOD Conference (SIGMOD'19).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SIGMOD'19, June 2019, Amsterdam, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Complex event processing (CEP) methods are widely employed in applications where arbitrarily complex combinations (patterns) of data items must be promptly and efficiently detected in massive data streams. Examples of such areas include financial services [22], electronic health record systems [14], sensor networks [32], and more recently IoT [66]. CEP systems treat data items as *events* arriving from event sources. As new events are detected, they are combined into higher-level *complex events* matching the user-specified patterns. An active area of academic research [5, 6, 19, 22, 49, 62], CEP functionality is also provided by multiple commercial data analysis platforms [9, 15, 29, 53].

Modern CEP engines are typically required to support efficient simultaneous tracking of hundreds to thousands of patterns in multiple high-speed input streams of events. We will refer to systems possessing this functionality as *multi-pattern complex event processing (MCEP) systems.*

As an example, consider a security system monitoring a corporate building. Every room entrance is equipped with a sensor that emits a signal to the main controller whenever any large object passes through the doorway. We are interested in detecting a scenario in which an intruder is detected near doorway A, then immediately passes through entrance B, and finally enters doorway C. This pattern can be formulated as a sequence of three events, each corresponding to getting a signal from sensors A, B, and C respectively. A real-life MCEP system could define multiple 'abnormal' paths inside the building and specify a dedicated pattern for each path.

Pattern matches in CEP systems are detected using an *evaluation mechanism.* One of the most prominent evaluation mechanisms is the non-deterministic finite automaton (NFA) [6, 23, 62]. Figure 1(a) presents an example of a NFA for detecting the sequence $A \rightarrow B \rightarrow C$ of sensor signals. A state

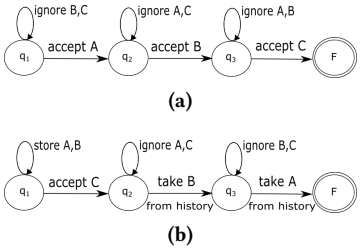

Figure 1: Evaluation mechanisms for a sequence of events from streams *A,B,C*: (a) NFA without reordering; (b) NFA with reordering.

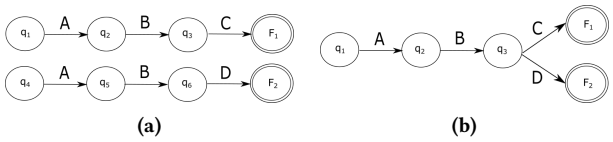

Figure 2: NFA sharing example for event sequences *A,B,C* and *A,B,D*: (a) no sharing; (b) prefix sharing is applied.

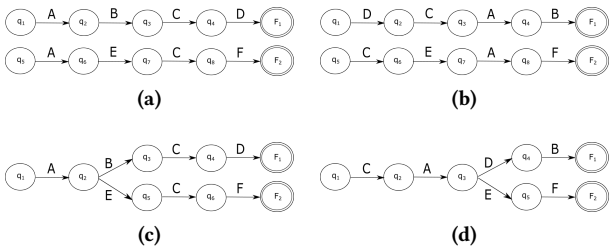

Figure 3: NFA optimization example for event sequences *A,B,C,D* and *A,E,C,F*: (a) no sharing or reordering; (b) reordering without sharing; (c) sharing without reordering; (d) a combination of reordering and sharing.

is defined for each prefix of a valid match. Every 'accepting' transition between states is associated with some event type. The detection is triggered by the arrival of a signal from sensor A. For each accepted signal, the stream of events from sensor B is probed. If a new signal is subsequently received from B, we wait for a corresponding event from sensor C.

During evaluation, a NFA keeps track of *partial matches*, that is, already detected subsets of a potential pattern match. A newly arrived event is combined with all currently stored partial matches corresponding to the state accepting this event. For instance, an event of type C will be matched with pairs of A's and B's. This architecture leads to the worst-case exponential (in the pattern size) processing time and memory consumption. Thus, maximizing pattern detection performance is crucial in a MCEP system.

Multiple research efforts have targeted various possibilities for creating efficient evaluation mechanisms. Two of the most popular optimization strategies are *pattern rewriting* and *pattern sharing* [33].

Pattern rewriting methods [7, 40, 49, 52, 57] exploit the statistical properties of the event data to replace the evaluation mechanism with an equivalent yet more efficient one. *Pattern reordering* is a more specific technique within this category, focused on modifying the order in which the events are processed. For example, let us assume that sensor C generates significantly fewer signals than A and B do. Then, instead of following the order $A \rightarrow B \rightarrow C$ specified by the pattern, it would be beneficial to first wait for a signal from C, then examine the local history for previous signals received from sensors B and A. This way, fewer partial matches would be created, resulting in better memory utilization and faster processing of incoming events [42]. Figure 1(b) depicts a NFA constructed according to this improved plan.

Pattern sharing methods [7, 22, 45, 54, 64] utilize the structural similarities between different patterns to unify the processing of common subexpressions. Figure 2 illustrates this principle. For presentational purposes, we omit 'ignore' edges and 'accept' labels from now on. We are required to monitor a pair of patterns $P_1 : A \rightarrow B \rightarrow C$ and $P_2 : A \rightarrow B \rightarrow D$. Instead of processing them independently (Figure 2(a)), the system can merge the first three states of the respective NFAs to produce a joint automaton (Figure 2(b)). This optimization avoids duplicate instantiating and storing of partial matches.

Pattern reordering and pattern sharing are generally considered as orthogonal techniques and cover different aspects of CEP performance optimization. This also implies that each of the two methods overlooks certain opportunities exploited by the other. Consequently, their fusion could discover evaluation plans that would not be considered otherwise.

We will illustrate the above using the following example. The system is given two patterns depicted in Figure 3(a). Reordering the patterns by the ascending order of event arrival rates might result in a pair of locally optimal NFAs (Figure 3(b)). Alternatively, a global shared plan shown in Figure 3(c) can be obtained by sharing the first two states. Now consider a combined application of the above techniques, where the NFAs are first reordered to maximize the common prefix length, and then this newly created subpattern is shared. Figure 3(d) shows the resulting plan. This plan would never be created if only one of the two optimizations was employed, or if they were used independently.

Despite the benefits of combining pattern sharing with rewriting for MCEP optimization, surprisingly little work has been done in this direction. The only related publications known to the authors either limit the discussion to a single shared subpattern [7] or consider a restricted model for pattern decomposition without reordering [64].

In this paper, we present a novel framework for large-scale MCEP. Rather than merely maximize the sharing degree or create locally optimal plans, we aim to produce a globally optimal plan for the given workload of patterns using a mixture of the two. At the core of our framework lies the optimizer that uses sharing and reordering techniques to generate candidate evaluation plans. This fusion allows us to take advantage of subexpressions not normally considered for sharing. To traverse the hyperexponential space of plans, we incorporate a method based on the local search paradigm [3]. As opposed to the traditional MCEP optimizers, our system can operate under arbitrarily tight time constraints due to the inherent balance between optimization time and solution quality. To the best of our knowledge, no existing work discusses this optimization strategy in a MCEP context.

Our contributions can thus be summarized as follows:

- We present a novel approach for optimizing large-scale MCEP systems by combining the power of state-of-the-art pattern sharing and reordering techniques.
- We design a set of algorithms for efficiently searching the solution space. Our algorithms are highly precise and their execution time can be arbitrarily limited.
- We implement a MCEP engine utilizing the plans created by our optimizer for efficient pattern detection.
- We empirically validate the performance of our solution, demonstrating its scalability and superiority over existing state-of-the-art approaches.

The remainder of this paper is organized as follows. Section 2 introduces the notations used throughout the paper. In Section 3 we describe our system design and present a limited version of our optimization mechanism, restricted to prefix sharing. Section 4 presents the algorithmic framework for selecting the best evaluation plan from the solution space. Section 5 extends the method from Section 3 to consider sharing of arbitrary subsets. We report the results of our experimental study in Section 6. Section 7 discusses the related work. Section 8 summarizes the paper.

2 BACKGROUND AND TERMINOLOGY

Formally, a MCEP system accepts three parameters: an input data stream I, a pattern workload WL, and a statistics collection $Stat$. The input stream $I = \{e_1, e_2, \cdots\}$ is an ordered, possibly infinite temporal sequence of *primitive events*, or simply events. We define I as a "logical" input source, possibly encapsulating multiple merged substreams. Each event e_i is represented by a well-defined type and a set of attributes, including the occurrence timestamp. In the example from Section 1, the event type is specified by the origin sensor ID, and the attribute set may include the movement speed of an intruder or the direction of passing.

The workload $WL = \{P_1, \cdots, P_n\}$ contains a finite number of patterns the system is requested to detect. Each pattern is defined by the tuple $P_i = (\mathcal{E}_i, S_i, C_i, W_i)$, where $\mathcal{E}_i = \{E_1, \cdots, E_{m_i}\}$ is the set of event types participating in P_i, S_i denotes the *structure* of P_i (which will be defined shortly), C_i is the *condition set* specifying the constraints on the attribute values of the events, and W_i is the *time window* defined for this pattern, that is, the maximal allowed time difference between the timestamps of a pair of events in a match.

The structure S_i specifies how the events requested by the pattern are to be assembled to form a match. It is defined by a combination of event types and operators. In this paper, we will consider the most common operators such as AND, SEQ, and OR. The AND operator requires the occurrence of all events specified in the pattern. The SEQ operator also expects the events to appear in a predefined temporal order. The OR operator corresponds to the appearance of any event out of those specified. Two additional important operators are the negation (NOT), requiring the absence of an event from some position in the match, and the Kleene closure (KL), accepting one or more instances of an event.

To illustrate the above, the structure of the pattern from Figure 1 could be summarized as $SEQ(A, B, C)$, with $\mathcal{E} = \{A, B, C\}$. If the order of receiving the signals was not important, the pattern would be formulated as $AND(A, B, C)$. In addition, assume that a signal arriving from the sensor D indicates the arrival of a security guard to the area, in which case no alarm should be set. Then, the structure of our pattern would become $SEQ(AND(A, B, C), NOT(D))$.

In the general case, S_i is an arbitrary expression over the above operators. As discussed in previous work [40, 41], such patterns can be simplified by the transition to DNF form. From the standpoint of a MCEP system, every clause of the resulting DNF expression can be considered as a separate pattern in a workload. In addition, a clause containing multiple AND/SEQ operators can be flattened to a simple expression featuring a single AND or SEQ with possible NEG and KL applied on single events [41]. Therefore, we will only consider patterns of this simplified form from now on.

$Stat$ is a set of statistical data properties that are used by the MCEP engine during evaluation plan generation. In the example above, $Stat$ contains the arrival rates of all event types (that is, of signals from each sensor). In addition, we will consider as members in $Stat$ the selectivities of the conditions defined by the patterns. The selectivity of a condition is defined as the probability of the input tuple to successfully pass the condition. More formally,

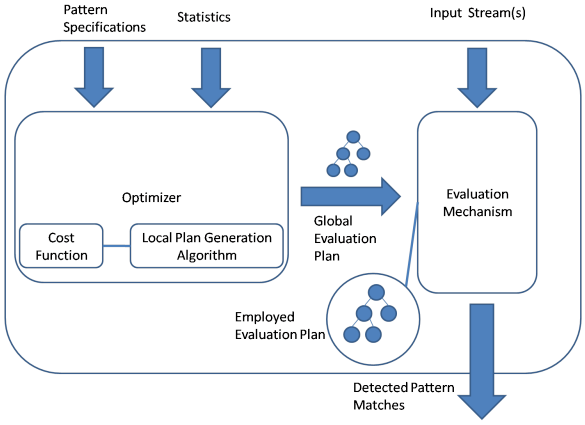

Figure 4: General structure of a MCEP system.

$$Stat = \{r_x | \exists P_i \in WL : E_x \in \mathcal{E}_i\} \cup \left\{sel^{C_i}_{x,y} | \exists P_i \in WL : E_x, E_y \in \mathcal{E}_i\right\},$$

where r_x is the arrival rate of the event type E_x, and $sel^C_{x,y} \in [0,1]$ is the selectivity of a mutual condition between E_x and E_y in some condition set C (we set $sel^C_{x,y} = 1$ if no condition is defined between the event types). Our results can be trivially extended to additional parameters, such as inter-event dependencies and costs of predicate evaluation, by modifying the cost model (see below).

The general architecture of a MCEP system is depicted in Figure 4. The *evaluation mechanism* is responsible for the actual processing of the input stream I. An evaluation mechanism of choice in Figures 1-3 is a NFA. Various works describe different variations of NFAs [6, 19, 22, 62]. In this paper, we exclusively use the 'lazy NFA' introduced in [42]. A lazy NFA (Figure 1(b)) can be configured to follow any execution order regardless of the actual order requested by the pattern. Since NFAs discussed in previous works are only capable of tracking a single pattern, an extension for multiple patterns will be presented in Section 3. A different evaluation mechanism will be introduced in Section 5.

At runtime, the evaluation mechanism follows an *evaluation plan* supplied by the optimizer. We distinguish between *local evaluation plans* applicable for single-pattern evaluation mechanisms only, and *global evaluation plans* that consider a workload of patterns. For example, the plans applied by the NFAs in Figures 1(a) and 1(b) are local evaluation plans, whereas Figures 2 and 3 illustrate global evaluation plans.

Different evaluation mechanisms support different types of evaluation plans. Creating a lazy chain-structured NFA (Figure 1) for a single pattern requires an *order-based* local evaluation plan. For a pattern P over the event types $E_1, \cdots, E_m$, the order-based evaluation plan is an ordering $O = (E_{q_1}, \cdots, E_{q_m})$, where $q_1, \cdots, q_m$ is a permutation of $[1, \cdots, m]$. As described in [41], any pattern using the operators defined earlier in this section (with the exception of OR) can be detected by such NFA. We will discuss the structure of global order-based evaluation plans later in the paper.

The task of the *optimizer* is to create a global evaluation plan upon system initialization. The resulting plan is then transferred to the evaluation mechanism, which subsequently launches the detection process on a stream I. The optimizer typically uses a predefined *cost function* to measure the quality of a plan subject to the given workload WL and the statistics collection $Stat$. We will define this function as $Cost : \mathfrak{P} \times \mathbb{W} \times STAT \rightarrow \mathbb{R}$, where $\mathfrak{P}$, $\mathbb{W}$,$STAT$ are the sets of all global evaluation plans, workloads, and statistics collections, respectively. The cost assigned by this function may reflect performance metrics such as throughput, detection latency, communication cost, and more.

Our analysis below assumes the values in $Stat$ to be constant and known in advance. However, in real-life scenarios this information is rarely obtained in advance and is subject to rapid fluctuations over time. To overcome this problem, our system employs standard adaptivity mechanisms [39, 42, 49], continuously estimating the up-to-date statistics and relaunching the optimizer when a significant change is detected.

3 MULTI-PATTERN CEP WITH PREFIX SHARING

In this section, we present the core principles and algorithms behind our MCEP system. For presentational purposes, we describe a limited version of our method, only considering prefix sharing opportunities between patterns. In Section 5, we extend our framework to support arbitrary subexpression sharing. The experimental study in Section 6 is conducted solely on this extended system.

3.1 Multi-Pattern NFA Evaluation

Our framework processes all patterns in a workload using a single NFA, which we denote as the *multi-pattern NFA*. It is organized in a tree-like topology formed by merging the common prefixes of the chain-structured NFAs corresponding to each pattern in the workload. The root of the tree is shared between all patterns and serves as the initial state of the automaton. Each internal node can be shared between two or more patterns.

Since different patterns may have different time windows, we augment each state of the multi-pattern NFA with a special *time window* attribute, set to the largest time window among the patterns sharing the state. The system uses this attribute to decide whether a partial match has expired.

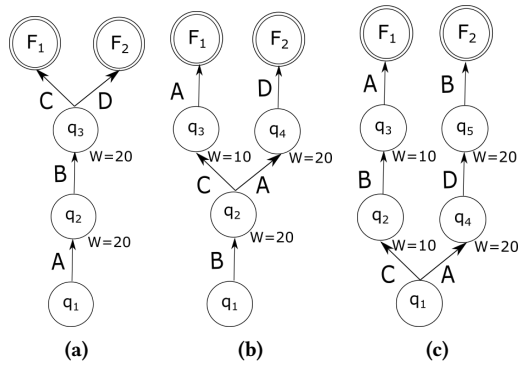

Figure 5: Multi-pattern trees for a workload consisting of *SEQ(A,B,C)* and *SEQ(A,B,D)*: (a) evaluation orders *A,B,C* and *A,B,D* (maximal sharing); (b) evaluation orders *B,C,A* and *B,A,D*; (c) evaluation orders *C,B,A* and *A,D,B* (minimal sharing).

Figure 5 depicts three of the possible multi-pattern NFAs for a workload of two patterns, $P_1 : SEQ(A, B, C)$ and $P_2 : SEQ(A, B, D)$, with $W_1 = 10$ and $W_2 = 20$. As discussed in Section 1, some NFAs have more shared states, while others contain more states in total but provide more efficient evaluation paths for individual patterns.

For each pattern in a workload, a dedicated *final state* is defined. When the final state corresponding to some pattern is reached, a match is reported. Note that while final states are typically the leaves of the tree, this is not always the case. For example, in a workload consisting of $SEQ(A, B, C)$ and $SEQ(A, B)$, the final state for $SEQ(A, B)$ is an internal node.

The evaluation process for multiple patterns is similar to the one presented in [42] for single-pattern detection. As a new event e of type T enters the system, it is evaluated against existing *NFA instances*. An instance is defined by a combination of a unique state identifier and a partial match. The system starts with a single instance associated with the initial state and an empty match. All instances associated with states containing an outgoing transition for T are matched with e. For every instance satisfying the conditions between the events (including e), a new instance is created containing the new match resulting from e's addition and associated with the state to which the transition leads. When an instance corresponding to some final state is created, its match is reported to the end users. An instance exceeding the time window specified by its associated state is removed from the system.

Since the number of instances in a system processing a large workload may be huge, traversing all of them on every event arrival is impractical. Instead, for each event type T we define a list l_T to contain all states with an outgoing transition accepting T. The size of l_T can never exceed the number of patterns in a workload containing T in their specification and will be substantially lower under an efficient sharing strategy that aims to merge states that process interleaving event types. At runtime, NFA instances are stored in a hash table according to their associated state, and the arrival of an event of type T only triggers the traversal of instances associated with states in l_T. For example, the state lists of a multi-pattern NFA in Figure 5(b) are $l_A = \{q_2.q_3\}, l_B = \{q_1\}, l_C = \{q_2\}, l_D = \{q_4\}$.

3.2 Multi-Pattern Tree

Global evaluation plans utilized by multi-pattern NFAs are similarly structured in a tree-like manner. We will refer to this plan type as the *multi-pattern tree (MPT)*. Given a MPT, a multi-tree NFA is constructed by simply copying the structure of the former.

As described in Section 2, a MPT is created by the optimizer. As we will see in Section 4, our optimizer proceeds by creating an initial MPT and repeatedly modifying it. Hence, efficient creation and modification operations are crucial for minimizing the optimization cost. In implementing these operations, the core principle of MPT behavior is to unconditionally share all shareable prefixes of the supplied local evaluation plans (orders). To add an evaluation order O to an existing MPT, we iterate over O and only create a new node if no equivalent one exists. Two nodes are considered equivalent if and only if they correspond to identical sequences of event types, and if their edges specify identical conditions. Similarly, a plan is removed by iterating over the respective order and only deleting states that are not shared with other patterns.

Figure 6 illustrates an addition and a removal of a plan from a MPT. The complexity of both operations is $O(m)$, where m is the length of the evaluation order.

Creating a MPT from a set of orders $\{O_1, \cdots, O_n\}$ is implemented by iteratively adding the orders to an initially empty tree. This operation requires $O(n \cdot max(m_i))$ time and space, where m_i is the length of O_i.

Since MPTs merge all common prefixes, we can uniquely define a MPT by the tuple $(O_1, \cdots, O_n)$. Forcing some nodes not to be shared is only possible by modifying the individual evaluation orders. This way, careful selection of local evaluation plans by the optimizer can achieve the perfect balance between sharing degree and local evaluation plan quality.

3.3 Runtime Complexity and Multi-Pattern Cost Model

We will now analyze the runtime complexity of the MCEP evaluation process described above and derive the cost function definition for multi-pattern trees.

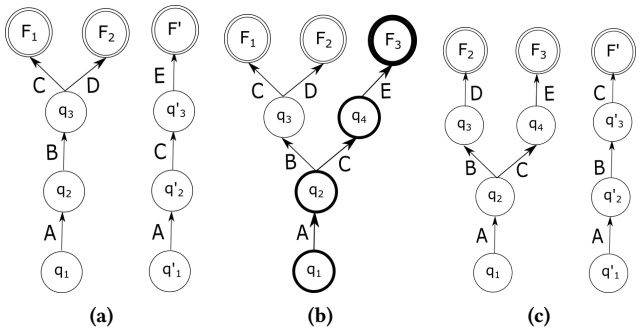

Figure 6: MPT modification example: (a) a MPT from Figure 5(a) and a local plan for a pattern *SEQ(A,C,E)*; (b) the MPT following the addition of the new evaluation plan (the path corresponding to the newly added plan is highlighted); (c) the MPT after the local evaluation plan for *SEQ(A,B,C)* is removed.

The total cost associated with processing a single event e of type T is the sum of two components: 1) the cost of combining e with the existing partial matches and creating new instances as a result of successful matching; 2) the cost of purging the instances created as a result of e's arrival upon their expiration. We will denote the former as $CP(T)$ and the latter as $CR(T)$.

Both functions depend on the expected number of instances active at the time of an event arrival. Reducing the number of instances (or, more generally, the size of intermediate results) is a common optimization goal in multiple fields, including database query optimization [18, 58, 61] and complex event processing [42, 49, 57]. In [40], a cost model was developed to estimate this metric for single-pattern lazy NFA evaluation. For an order-based plan $O = (E_{q_1}, \cdots, E_{q_m})$ detecting a pattern $P = (\mathcal{E}, S, C, W)$, this cost function is defined as:

$$Cost_{ord}(O, P, Stat) = \sum_{k=1}^{|\mathcal{E}|} Cost^k_{ord}(O, P, Stat),$$

where $Cost^k_{ord}$ is the cost of the k^{th} state in the chain-based NFA following O, calculated as follows:

$$Cost^k_{ord}(O, P, Stat) = W^k \cdot \prod_{i=1}^{k} r_{q_i} \cdot \prod_{i,j \le k; i \le j} sel^C_{q_i, q_j},$$

where r_i; $i \in [1, m]$ and $sel^C_{i,j}$; $i, j \in [1, n]$ are as defined in Section 2.[1]

[1]The presented function is the basic version of $Cost_{ord}$ and it only applies when no negation or Kleene closure operator appears in the pattern. The reader is referred to [40] for the full definition of the cost function.

We will use the above definition to calculate the expected number of instances existing simultaneously at any given moment during MPT-based multi-pattern evaluation. Given a node N, let $\mathcal{P}_N$ denote the path from the root of the MPT to N (by definition of a tree, there is always exactly one such path). For the root, we set $\mathcal{P}_R = \varnothing$. The total number of instances is the sum of numbers of instances associated with each NFA state (and hence with the corresponding MPT node), calculated as follows:

$$\#inst(MPT, WL, Stat) = \sum_{N \in MPT} Cost^{|\mathcal{P}_N|}_{ord}(\mathcal{P}_N, WL, Stat).$$

Thus, to calculate the number of instances to be traversed upon arrival of an event of type T, we need to sum the instances associated with the states in l_T:

$$\#inst_T(MPT, WL, Stat) = \sum_{S \in l_T} Cost^{|\mathcal{P}_{N(S)}|}_{ord}\left(\mathcal{P}_{N(S)}, WL, Stat\right),$$

where $N(S)$ denotes a node corresponding to S in MPT.

The processing cost per event is now derived as follows. Let C_a be the cost of accessing an instance, C_n the cost of creating a new instance and inserting it into the data structure, and C_r the cost of removing an instance from the system. In addition, let $C_v(T, \mathcal{P}_N)$ denote the cost of verifying the conditions between a new event of type T and the events preceding T in $\mathcal{P}_N$, and let $Sel_v(T, \mathcal{P}_N)$ denote the total selectivity of the above conditions. To make C_v and Sel_v well-defined, we set $C_v = Sel_v = 0$ if $T \notin \mathcal{P}_N$. Then, the expected cost of processing a single event of type T is:

$$CP(T) = \sum_{S \in l_T} \Big(Cost^{|\mathcal{P}_{N(S)}|}_{ord}\left(\mathcal{P}_{N(S)}, WL, Stat\right) \cdot \left(C_a + C_v\left(T, \mathcal{P}_{N(S)}\right) + Sel_v\left(T, \mathcal{P}_{N(S)}\right) \cdot C_n\right) \Big).$$

To calculate the cost of removing the expired instances, we observe that the expected number of instances created in state S after processing a new event of type T is equal to $Sel_v\left(T, \mathcal{P}_{N(S)}\right)$. Thus, the cost of eventually removing these instances upon their expiration is:

$$CR(T) = \sum_{S \in l_T} Cost^{|\mathcal{P}_{N(S)}|}_{ord}\left(\mathcal{P}_{N(S)}, WL, Stat\right) \cdot Sel_v\left(T, \mathcal{P}_{N(S)}\right) \cdot C_r.$$

The above analysis emphasizes two main performance objectives of a MCEP system attempting to minimize the processing cost per event. First, the sharing degree needs to be maximized to reduce the sizes of the state lists l_T. Second, the cost of the local evaluation plans in terms of the expected number of simultaneously existing instances has to be as low as possible. As illustrated in Figure 3, there might be a conflict between these two objectives, which we will solve by defining an optimization problem later on.

The extended formula for the expected number of instances represents the same parameter dependencies as does the expression $CP(T) + CR(T)$. Hence, we will use it as our cost function for measuring the quality of MPTs:

$$Cost_{ord}^{multi}(MPT, WL, Stat) = \#inst(MPT, WL, Stat).$$

3.4 MCEP Optimization Problem

We will now formally define the problem to be solved by the MCEP optimizer. Given an order-based plan O for a pattern P and a multi-pattern tree MPT, we say that $O \in MPT$ if and only if MPT contains a path $\mathcal{P}$ of length $|O|$, starting at the root and ending at some final state, such that the event types and the conditions specified on the transitions in $\mathcal{P}$ are identical to those of a NFA detecting P according to O. For example, a MPT in Figure 6(b) satisfies $O_3 = (A, C, E) \in MPT$. Also, we will denote by ORD_P the set of all valid order-based evaluation plans for P. For a pattern of size m, $|ORD_P| = m!$.

We are now ready to define our optimization problem.

Tree-based MCEP optimization problem (T-MCEP). Given a workload WL of n patterns and a statistics collection $Stat$, find a multi-pattern tree MPT minimizing the value of the cost function $Cost_{ord}^{multi}(MPT, WL, Stat)$ subject to

$$\forall P_i, 1 \leq i \leq n : \exists O \in ORD_P \text{ s.t. } O \in MPT.$$

We will denote the path in the MPT corresponding to the evaluation order of a pattern P_i as $\mathcal{P}_i$.

We will now discuss the complexity of T-MCEP. It can be noted that for $n = 1$ our problem is equivalent to the single-pattern CEP optimization problem (SCEP), thoroughly discussed in previous work [40, 49, 52, 57]. In particular, it was shown in [40] that SCEP is NP-complete by reducing it to the problem of join evaluation order generation. The NP-completeness of this latter problem was in turn proven by [18, 37] through a reduction to the maximum clique problem. The maximum clique problem is not only known to be NP-complete, but is also hard to approximate. It was demonstrated in [31] that, unless $NP = ZPP$, no polynomial-time algorithm exists that approximates the problem within the factor of $n^{1-\varepsilon}$, where n is the size of the graph. By correctness of the reductions, this result applies also to the SCEP problem, and, by generalization, to T-MCEP.

4 OPTIMIZATION FRAMEWORK FOR T-MCEP

T-MCEP is a computationally hard optimization problem, characterized by an enormously large solution space and multiple local minima. Therefore, advanced techniques are needed in order to produce a high-quality solution under tight restrictions common for real-time MCEP systems.

The algorithms employed by our optimizer to achieve this goal implement the local search paradigm [3, 36]. Local search is a well-known approach for finding approximate solutions for hard optimization problems, based on executing heuristically guided random walks in the solution space and searching for the cheapest solution subject to a predefined cost function. Local search methods are successfully applied for solving a wide range of problems, from the classic traveling salesman problem to code design and VLSI layout synthesis [3].

To the best of our knowledge, no prior work attempted to represent the task of stream or event processing optimization as a local search problem. Instead, related research efforts focused on utilizing heuristic approaches [56], dynamic programming [7, 49], local-ratio approximation algorithms [54], and branch-and-bound methods [28, 64] for similar optimization problems with hyperexponential solution spaces.

Local search methods present several important benefits for real-time streaming applications, and in particular for MCEP. Most importantly, they offer a tradeoff between the quality of the returned solution and the running time of the search. Since the local search procedure keeps a "current best" solution at any point of its execution, it can always be interrupted due to expired time limit and will return a valid solution, albeit not necessarily the cheapest. This property makes local search methods an attractive choice for targeting the MCEP optimization problem under tight real-time constraints.

We start this section by describing a data structure for managing inter-pattern sharing opportunities, which we denote as a *multi-pattern graph* (MPG). We then present a set of algorithms utilizing the MPG and implementing the local search paradigm to solve T-MCEP.

4.1 Multi-Pattern Graph

We will start with some preliminary definitions. Let $\pi_{\mathcal{X}}(Y)$ denote a projection of an expression Y on a set of variables $\mathcal{X}$. Y can be either a pattern structure or a condition set as defined in Section 2. For example, $\pi_{\{B,D\}}(SEQ(A, B, C, D)) = SEQ(B, D)$. Given a pattern $P = (\mathcal{E}, S, C, W)$, we will say that another pattern $P' = (\mathcal{E}', S', C', W')$ is a subpattern of P (marked as $P' \subseteq P$) if $\mathcal{E}' \subseteq \mathcal{E}$, $S' = \pi_{\mathcal{E}'}(S)$, $C' = \pi_{\mathcal{E}'}(C)$, and $W' \leq W$.

A common subpattern $P_{ij} = (\mathcal{E}_{ij}, S_{ij}, C_{ij}, W_{ij})$ of two patterns P_i, P_j is a pattern satisfying $(P_{ij} \subseteq P_i) \wedge (P_{ij} \subseteq P_j)$, such that $W_{ij} = min(W_i, W_j)$. A maximal common subpattern of P_i, P_j is a common subpattern P_{ij}, such that no other common subpattern P'_{ij} satisfies $P_{ij} \subseteq P'_{ij}$. We will denote it by MP_{ij} from now on. In addition, we will denote by Γ_{ij} the set of all subsets of MP_{ij}, that is, all common subpatterns of P_i and P_j. Obviously, $\Gamma_{ij} = \Gamma_{ji}$ for each i, j. The above definitions are trivially extended to an arbitrary number of intersecting patterns.

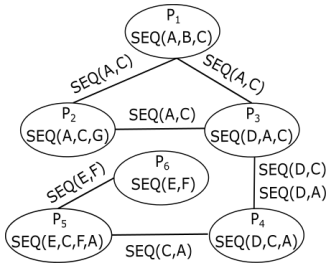

Figure 7: A multi-pattern graph for a workload of 6 patterns. Edges corresponding to maximal common subpatterns of size 1 are not shown. The triplet P_1, P_2 and P_3 shares a maximal common pattern $SEQ(A, C)$. P_3 and P_4 have two distinct maximal common subpatterns. P_6 is fully contained in P_5.

To illustrate the above notations, let $P_1 : SEQ(A, B, C, D)$ and $P_2 : SEQ(A, E, C, D)$. Assume that both patterns have no conditions and $W_1 = 10, W_2 = 20$. Then, $SEQ(A, D)$, $SEQ(C, D)$, and $SEQ(A, C)$ with $W = 10$ are common subpatterns of P_1 and P_2, while $SEQ(C, A)$ is a subpattern of neither, since it has a conflicting structure. The maximal common subpattern is $SEQ(A, C, D)$.

The *multi-pattern graph* $MPG = (V, E)$ is a data structure capable of efficiently collecting, maintaining, and retrieving the information regarding the mutual subpatterns of $P_1, \cdots, P_n$. For each pattern P_i, MPG contains a vertex $v_i \in V$. For each pair of distinct patterns P_i, P_j with non-empty intersection (i.e., satisfying $\Gamma_{ij} \neq \varnothing$), an undirected edge $e_{ij} = (v_i, v_j, \Gamma_{ij}) \in E$ is defined.

Figure 7 depicts a MPG for a workload of 6 patterns. For presentation clarity, edges with maximal common subpattern of size 1 are not shown.

In the general case, a MPG is an arbitrary, not necessarily connected graph. However, it can be noted that any algorithm solving T-MCEP can be activated separately on each connected component, and the results can then be combined to produce the final plan. Not only does this observation allow us to solve the problem much more efficiently in the presence of multiple components, but it also makes it possible to limit the discussion below to connected graphs.

To guarantee an efficient local search procedure, the MPG has to occupy small space. Moreover, addition and removal operations must be fast and low-cost, and likewise for the retrieval of pattern intersection information. By utilizing compact graph representation and advanced optimizations, we are able to guarantee near constant cost of retrieval and worst-case linear cost of addition and deletion with near linear space complexity. The techniques for optimizing MPG performance are described in detail in Appendix A.

4.2 Local Search Algorithms for T-MCEP

A local search problem is specified by a pair (φ, f), where φ is a set of feasible problem solutions and $f : \varphi \rightarrow \mathbb{R}$ is a cost function. The goal is then to find an optimal solution $s*$ such that $f(s*) \leq f(s)$ for all $s \in \varphi$. In the case of T-MCEP, φ consists of all possible MPTs and $f \equiv Cost^{multi}_{ord}$.

The search starts from some initial solution s_{init}. Local search algorithms traverse the search space by exploring the *neighborhood* of the current solution. A domain-specific *neighborhood function* $\mathcal{N} : \varphi \rightarrow 2^{\varphi}$ maps a solution to a set of its neighbors, i.e., solutions that can be obtained by performing a slight modification. The strategy for performing the search is determined by the *meta-heuristic* in use. A local search algorithm for a given problem can be uniquely defined by a combination of a meta-heuristic and a neighborhood function. When a predefined stopping criterion is satisfied, the search terminates and the cheapest observed solution is returned.

The local search algorithms employed by our optimizer for solving T-MCEP utilize two well-known meta-heuristics, *simulated annealing* and *Tabu search*. Appendix B provides the background on these methods and outlines our implementation choices. The remainder of this section focuses on our problem-specific neighborhood functions utilizing the information in the MPG to create candidate solutions.

It can be noted that the solution space of our problem is enormously large. For a workload of size n, there are $\prod_{i=1}^{n} |P_i|!$ possible MPTs, where $|P_i|$ denotes the number of event types in the i^{th} pattern. Fortunately, closer analysis of the solution space will allow us to immediately discard the overwhelming majority of the subplan combinations.

We can observe the following regarding the possible local evaluation orders for a pattern P_i in the shared workload. If no subset of P_i can be shared with other patterns, it only makes sense to select the most efficient evaluation order. Otherwise, for every shareable subpattern $P' \subseteq P$, we have to consider an order that starts with the best order O' for P', then continues with the best order for the remainder of the pattern given O' as the prefix. Note that not only the maximal common subpatterns but also their subsets must be considered, including the empty subset (which is equivalent to the case when no such P' exists).

We will formally state the above in the following theorem.

Theorem 4.1. *Let MPT_{opt} be the optimal multi-pattern tree for some workload W. Then, for each path $\mathcal{P}_i$ in MPT_{opt} corresponding to the pattern P_i at least one of the following holds: (1) $\mathcal{P}_i$ is the optimal evaluation order for P_i; (2) $\mathcal{P}_i$ can be divided into a non-empty prefix $Pref_i$ that is shared with at least one additional pattern and a non-shared suffix Suf_i, and it is the most efficient local evaluation order for P_i out of those starting with $Pref_i$.*

The proof is straightforward by assuming that neither (1) nor (2) hold and showing that MPT_{opt} can be improved by modifying $Suff_i$ to make $\mathcal{P}_i$ the most efficient order starting with $Pref_i$, which contradicts the optimality of MPT_{opt}. Since $Suff_i$ is not shared by definition, improving it necessarily leads to an improvement of MPT_{opt}.

Theorem 4.1 reduces the maximal number of potential orders for a single pattern from $|P_i|!$ to $\sum_{j=1}^{n} |\Gamma_{ij}|$. However, to apply the above strategy, an algorithm is required to calculate local evaluation plans as described above. We will assume the existence of a deterministic *local plan generation algorithm* $\mathcal{A}$, capable of the following functionality:

(1) Given a pattern P and the statistical event characteristics $Stat$, return the cheapest local order-based evaluation plan O subject to $Cost_{ord}$.
(2) Given a pattern P, its subpattern P', an evaluation plan O' for P', and the statistics collection $Stat$, return the cheapest (subject to $Cost_{ord}$) local order-based evaluation plan O starting with prefix O'.

Many algorithms answering the above requirements have been proposed [7, 40, 42]. In particular, any greedy algorithm or an algorithm based on dynamic programming satisfies both conditions. While most algorithms are not guaranteed to produce an optimal result due to the NP-hardness of local evaluation plan generation [40], they provide empirically accurate approximations. In the example that we discussed in Section 1, $\mathcal{A}$ is an algorithm arranging the event types in the ascending order of their expected arrival frequencies.

With the above observation in mind, we will now define neighborhood functions for T-MCEP. The first function produces a neighboring solution by selecting a random edge (v_i, v_j) in the MPG and a common subpattern $P \in \Gamma_{ij}$. We restrict P to be different from the subpattern that is shared between P_i and P_j in the current MPT (however, its subpatterns are allowed). A neighbor will be generated by invoking $\mathcal{A}$ to create new evaluation orders O_i, O_j sharing a common prefix O_P, and replacing $\mathcal{P}_i, \mathcal{P}_j$ with the resulting orders. We will denote this neighborhood as an *edge-based neighborhood* and use the notation $\mathcal{N}_{edge}$ to refer to it. $\mathcal{N}_{edge}(MPT)$ will denote the set of all solutions that can be obtained by the above procedure. The size of the neighborhood produced by $\mathcal{N}_{edge}$ is $\frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1;j\neq i}^{n} |\Gamma_{ij}|$.

The main drawback of $\mathcal{N}_{edge}$ is that it can only attempt pairwise sharing. In many real-life scenarios, a single subexpression might be shared between patterns comprising a large fraction of the workload. While sharing such subexpression between all involved patterns may dramatically increase the performance, only considering two of them may fail to produce an improvement over the plan not sharing the expression at all. As a result, the sharing opportunity may be missed.

To overcome this limitation, we define *vertex-based neighborhood* $\mathcal{N}_{vertex}$ as follows. Let $V_i = \bigcup_{(v_i,v_j)\in E} P_{ij}$ be called the *vicinity* of v_i. Instead of an edge, the neighborhood function will select a vertex v_i and a subpattern P in the vicinity of v_i. Then, let Γ_P denote a set of all patterns containing P. This set can be efficiently retrieved from the MPG as described in Appendix A. We will select $min(k, |\Gamma_P|)$ patterns, where $k \geq 2$ is a predefined parameter. Then, $\mathcal{A}$ will be invoked to generate new evaluation orders sharing a common prefix O_P. We will denote the variation of $\mathcal{N}_{vertex}$ using a particular value for k as $\mathcal{N}^k_{vertex}$. Note that $\mathcal{N}^2_{vertex}$ is equivalent to $\mathcal{N}_{edge}$. The size of the neighborhood of $\mathcal{N}^k_{vertex}$ is bounded by $\sum_{i=1}^{n} \sum_{P\in V_i} \binom{|\Gamma_P|}{k}$.

The per-step complexity of the neighborhood functions $\mathcal{N}_{edge}$ and $\mathcal{N}^k_{vertex}$ is $O\left(\sum_{i=1}^{n} m_i \cdot O\right)$, where O is the complexity of $\mathcal{A}$. A step is defined as a single selection of a neighbor and evaluating its cost.

In all algorithms, the initial state is set to the MPT in which all patterns are evaluated according to the best possible local evaluation orders, that is, $\mathcal{P}_i = \mathcal{A}(P_i, Stat)$ for all i.

5 MCEP WITH ARBITRARY SUBEXPRESSION SHARING

The multi-pattern plan generation method in Section 3 only considers prefix sharing. This introduces a significant limitation, since the optimizer is required to move common subpatterns to the MPT root in order to share their computation. This mechanism also prevents a pattern from sharing multiple distinct subexpressions with other patterns. As an example, consider a workload consisting of patterns $P_1 : SEQ(A, B, C, D)$, $P_2 : SEQ(A, E, C, F)$, $P_3 : SEQ(G, B, H, D)$. In order to share the subpattern $SEQ(A, C)$ with P_2, the evaluation order of P_1 must start with (A, C) or (C, A). On the other hand, it has to start with (B, D) or with (D, B) to share the subpattern $SEQ(B, D)$ with P_3. The optimizer will have to refrain from sharing one of the subpatterns in this case.

In this section, we extend our optimization framework to arbitrary subexpression sharing. To that end, we replace the local order-based plans with *tree-based plans*, shaped as binary trees. Tree-based plans, first described in [49], specify the structure for tree-based single-pattern evaluation mechanisms. A leaf is defined for each event type, and the root of the tree serves as a final state. The evaluation proceeds from the leaves towards the root, with each internal node responsible for a subpattern consisting of the event types in its subtree. Figure 8 presents three possible tree-based plans for a pattern $SEQ(A, B, C)$. Tree-based evaluation mechanisms were shown by multiple studies to be more expressive and perform better than NFAs [39, 40, 49].

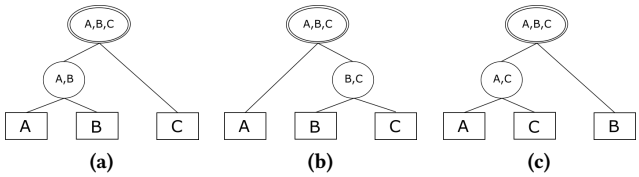

Figure 8: Tree-based plans for a pattern *SEQ(A,B,C).*

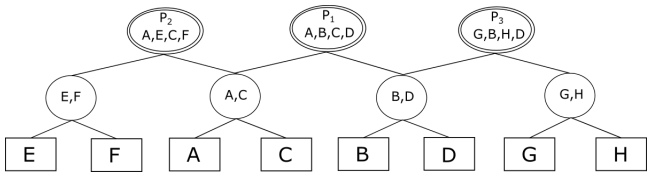

Figure 9: A multi-pattern multitree for a shared workload of patterns $P_1 : SEQ(A, B, C, D)$, $P_2 : SEQ(A, E, C, F)$, and $P_3 : SEQ(G, B, H, D)$.

The tree-based evaluation process is similar to the one described for NFAs. As a new event arrives, an instance is created containing this event. Every instance corresponds to some subtree s of the tree-based plan. A new instance I is combined with previously created "siblings", that is, instances associated with a node sharing the parent with the node of I. As a result, another instance containing the unified subtree is generated. This process continues iteratively until the root of the tree is reached or no siblings are found.

Similarly to MPT, we define a *multi-pattern multitree (MPM)* as the global plan consisting of multiple shared tree-based plans. Each pattern in a MPM has a dedicated root, and all leaves corresponding to the same event type are shared regardless of the plan in use. Figure 9 depicts a possible MPM for the example above. Note that the displayed plan successfully shares both subpatterns of P_1 with P_2 and P_3, a result that could not be achieved using an order-based approach.

The multitree-based MCEP optimization problem (M-MCEP) will be defined similarly to T-MCEP. The formal definitions of M-MCEP, the new cost functions $Cost_{tree}$ and $Cost^{multi}_{tree}$, and the corresponding extension of Theorem 4.1 can be found in Appendix C.

The MPM is created and modified similarly to the MPT. The complexity of the operations is not altered by switching to tree-based plans, as the number of nodes in a local tree-based plan is still linear in the number of the participating event types. In addition, the existence of a subtree T in a MPM can be tested in constant time (and an additional $O\left(\sum_{i=1}^{n} m_i\right)$ space) by hashing the subtrees upon creation. The complexity analysis of runtime evaluation from Section 3.3 also remains unchanged for the multitree model, with the exception of the cost function $Cost^{multi}_{ord}$ being replaced with $Cost^{multi}_{tree}$.

The local search process for MPMs functions as described for MPTs in Section 4.2. However, now it is possible for a pattern to share multiple disjoint subtrees. Consider a situation where one such subpattern $\hat{P}_1$ is already shared, and the optimizer attempts to share the second subpattern $\hat{P}_2$ during the local search step. In this case, we would like to consider two separate options: 1) the most efficient tree containing $\hat{P}_2$ regardless of the existing sharing of $\hat{P}_1$; 2) the most efficient tree containing both $\hat{P}_1$ and $\hat{P}_2$. This case can be generalized to sharing q subtrees and considering the $(q+1)^{th}$ one. Due to this extension, $\mathcal{A}$ is required to support multiple subtrees. More formally, we require $\mathcal{A}$ to be capable of the following:

(1) Given a pattern P and the statistical event characteristics $Stat$, return the cheapest local tree-based evaluation plan T subject to $Cost_{tree}$.
(2) Given a pattern P, a set of tree-based plans Υ for some subpatterns of P, and the statistics collection $Stat$, return the cheapest (subject to $Cost_{tree}$) local tree-based evaluation plan T containing all trees in Υ as subtrees.

Algorithms for tree-based plan generation satisfying the above requirements are discussed in [40, 49].

When selecting the next state to be returned, our neighborhood functions will randomly choose whether existing shared subtrees should be preserved for the patterns involved. For $\mathcal{N}^k_{vertex}$, this decision will be performed independently for each of the k patterns sharing a common subpattern. To apply this modification, we need to store sharing information for each pattern in the MPG, which adds a memory requirement of $O\left(n \cdot max_i\left(|\mathcal{E}_i|\right)\right)$. No further changes to the structure and the operations of the MPG are necessary for the tree-based evaluation model.

6 EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation. We assess the overall system performance achieved by our approach as compared to the state-of-the-art methods for MCEP, and analyze the impact of the various parameters on the quality of the generated global plans.

6.1 Experimental Setup

Two independent datasets were used in the experiments. The first was taken from the NASDAQ stock market historical records [1]. Each data record represents a single update to the price of a stock, spanning a 1-year period and covering over 2100 stock identifiers with prices periodically updated. Our input stream contained 80,509,033 primitive events, each consisting of a stock identifier, a timestamp, and a current price. We also augmented the event format with the precalculated difference between the current and the previous price of each stock. We considered updates of each stock identifier as events belonging to a separate type.

The structure of the patterns in the workloads generated for this dataset was motivated by the problem of monitoring the relative changes in stock prices. Each pattern represented either a sequence or a conjunction of a number of event types and included a number of predicates, roughly equal to half the pattern size, comparing the *difference* attributes of two of the involved event types. In addition, about 20% of the patterns contained either a negation or a Kleene closure operator on some event type. As discussed in Section 2, the aforementioned combinations of pattern operators are sufficient to cover the whole spectrum of pattern structures. For example, a typical sequence pattern of size 3 is as follows: $P_1 : SEQ(MSFT, Kleene(GOOG), APPL); C_1 = \{MSFT.diff < APPL.diff\}$.

The second dataset contains the vehicle traffic sensor data, provided by the city of Aarhus, Denmark [8] and collected over a period of 4 months from 449 observation points, with 13,577,132 primitive events overall. Each event represents an observation of traffic at the given point. The attributes of an event include, among others, the point ID, the average observed speed, and the total number of observed vehicles during the last 5 minutes. The patterns created for this dataset followed the rules specified above and were motivated by normal driving behavior, where the average speed tends to decrease with the increase in the number of vehicles on the road. We requested to detect the violations of this model, that is, combinations of three or more observations with either an increase or a decline in both the number of vehicles and the average speed.

Unless stated otherwise, all arrival rates and predicate selectivities were calculated in advance during the preprocessing stage. The measured arrival rates varied between 2 and 47 events per second, and the selectivities ranged from 0.003 to 0.92.

The workloads were created by grouping the patterns generated as described above based on a set of parameters, including the number of patterns in a workload, average pattern size (number of event types in a pattern), and pattern time window. Unless stated otherwise, the default values were set to 100 patterns per workload, an average pattern size of 5 event types, and the time window of 15 minutes. Experiments considering two additional parameters of interest are discussed in Appendix D.

Unless stated otherwise, all experiments were conducted on the full version of our MCEP optimizer presented in Section 5. The default local search time limit for all algorithms was set to 180 seconds. We used the algorithm based on dynamic programming described in [40] as our local plan generation algorithm $\mathcal{A}$.

We selected throughput, defined as the number of events processed per second during pattern detection, as our main performance metric. We believe that similar results could be obtained for algorithms targeting any other optimization goal, such as minimizing latency, power consumption, or communication cost.

All experiments were repeated on 10 independently generated workloads, and the displayed results were averaged among all trials. All models and algorithms were implemented in Java. The experiments were run on a machine with 2.20 Ghz CPU and 16.0 GB RAM.

6.2 Experimental Results

6.2.1 Impact of Input Parameters on System Performance. In our first experiment, we evaluated the performance of the local search algorithms described in Section 4 as a function of the workload size (Figure 10). Here and in all subsequent experiments, the graphs show the relative throughput gain over the trivial global evaluation plan, utilizing no sharing and no rewriting techniques. The neighborhoods $\mathcal{N}_{edge}$, $\mathcal{N}^4_{vertex}$, and $\mathcal{N}^8_{vertex}$ were tested in conjunction with simulated annealing and Tabu search meta-heuristics on stock (Figures 10(a)-10(b)) and traffic (Figures 10(c)-10(d)) datasets. For $\mathcal{N}_{edge}$ alone, the prefix-only version of our framework (Section 3) was evaluated in addition to the default arbitrary-subset version.

Overall, all combinations demonstrated more significant throughput gains for larger workloads, ranging from a factor of 21 to over 72. Despite being the simplest, $\mathcal{N}_{edge}$ neighborhood showed the best results, finding evaluation plans that outperformed the trivial plan by a factor of up to 72.7 for the stock dataset and up to 50.7 for the traffic dataset. This can be explained by the overwhelming size of the neighbor spaces explored by $\mathcal{N}^4_{vertex}$ and $\mathcal{N}^8_{vertex}$. Tight time constraints prevent the system from locating the best optimization opportunities in huge neighborhoods. Thus, although $\mathcal{N}_{vertex}$ neighborhoods contain all of the moves in $\mathcal{N}_{edge}$, the better moves are statistically harder to reach before the time expires. Comparable performance was observed for both meta-heuristics, with simulated annealing slightly outperforming Tabu search for the stock dataset and vice versa for the traffic dataset.

The choice of a subexpression sharing strategy was found to have a major impact on the system performance. When the optimizer was restricted to only consider sharing prefixes, applying the generated plans resulted in up to 5 times lower throughput (marked as 'EDGE-PREFIX' in all graphs) as compared to the plans produced using an identical setup without the above limitation (marked as 'EDGE'). This observation fully matches our prior analysis. As we discussed in Section 5, a prefix-only approach ignores a significant fraction of the space of possible optimizations and limits a pattern to only sharing a single subexpression by utilizing order-based local plans as opposed to tree-based ones.

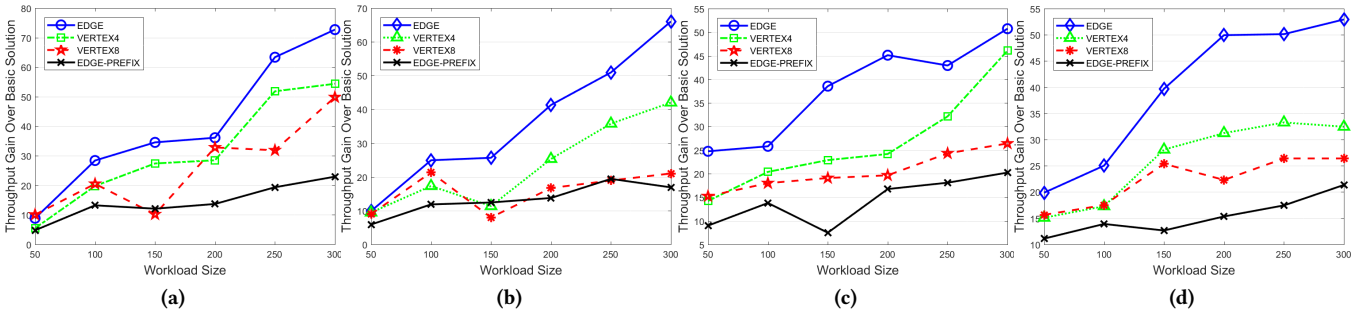

Figure 10: Throughput gain as a function of the workload size for different combinations of a meta-heuristic, a neighborhood function, a subexpression sharing strategy, and a dataset: (a) stocks dataset, simulated annealing; (b) stocks dataset, Tabu search; (c) traffic dataset, simulated annealing; (d) traffic dataset, Tabu search.

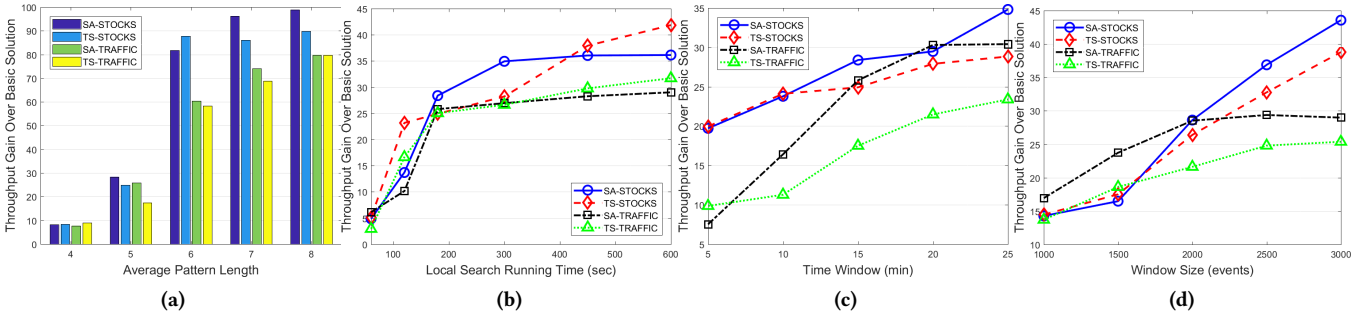

Figure 11: Throughput gain of the local search algorithms as a function of: (a) average pattern size; (b) local search running time; (c) pattern timestamp-based window; (d) pattern count-based window.

We further assessed the scalability of our optimizer subject to various parameters (Figure 11). Simulated annealing (marked as 'SA' in the graph) and Tabu search (marked as 'TS') were again evaluated on both datasets in conjunction with the best-performing neighborhood $\mathcal{N}_{edge}$. Figure 11(a) depicts the throughput gain as a function of the average length of a pattern in a workload. Our approach seems to improve even more for longer patterns, speeding up the event processing by up to two orders of magnitude. This is not surprising, as longer patterns introduce more optimization opportunities. We also observed that in most cases the simulated annealing meta-heuristic achieved better performance than Tabu search.

Unsurprisingly, the output plan quality also improves with increased time limit of the local search algorithm (Figure 11(b)). Interestingly, the performance of simulated annealing seems to converge to a constant value, while Tabu search keeps improving for longer time limits. This can be explained by the distinctive behavior of the former after a large number of iterations, when the current threshold becomes small enough for the algorithm to converge to a local minimum.

The results obtained for different time window sizes (Figure 11(c)) demonstrate similar trends. Since our cost function and the overall system throughput strictly depend on this parameter, increasing it leads to bigger differences in plan qualities, both calculated and empirically observed.

Finally, we experimented with patterns utilizing *count-based windows*. As opposed to specifications based on *time-based windows* that we defined in Section 2, count-based patterns require a match to appear within the last W arrived events rather than within W time units.

Figure 11(d) presents the results. For the stock dataset, even bigger performance boost was observed for larger windows as compared to the time-based scenario. This can be explained by the highly fluctuating event arrival rates exhibited by this dataset. When time-based windows are used, the peak load is only experienced during brief 'bursts', whereas large count-based windows cause the system to be constantly overloaded. Since the performance gain achieved by an efficient evaluation plan is proportional to the average system load, the latter case demonstrates a more significant increase in total throughput. In contrast, the results for the traffic

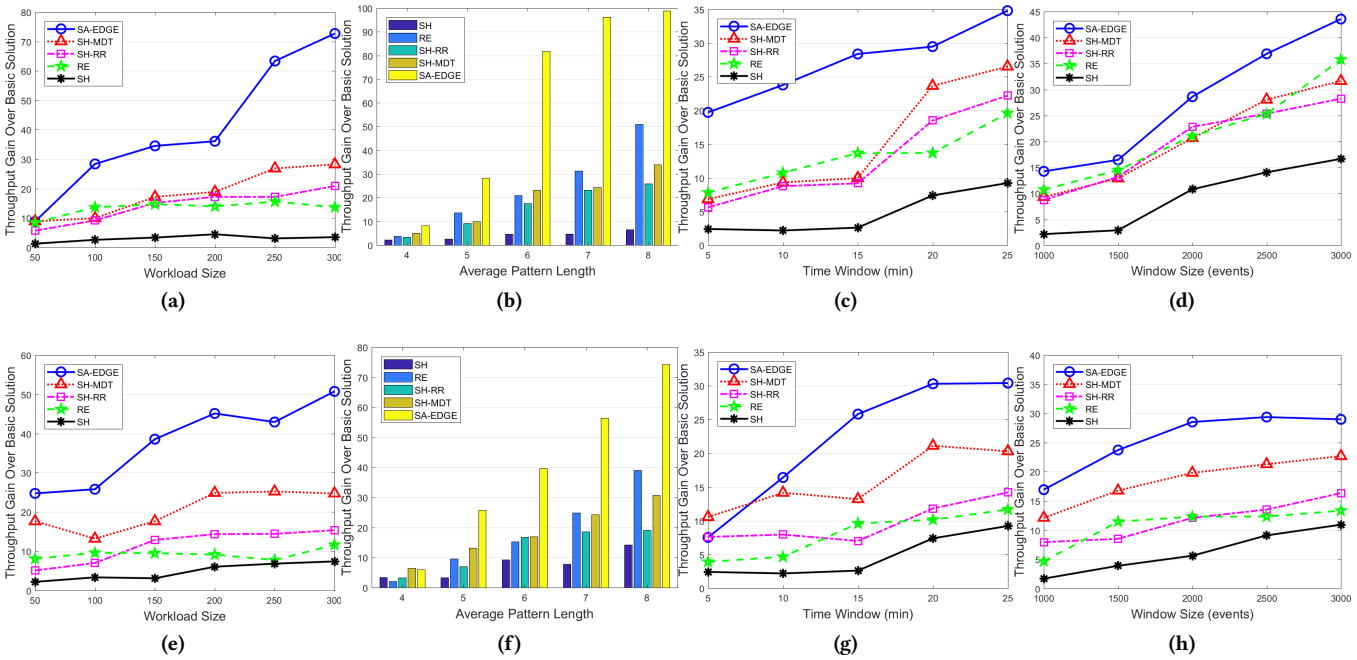

Figure 12: Throughput gain of the state-of-the-art and the local search algorithms applied on the stock dataset ((a)-(d)) and on the traffic dataset ((e)-(h)) as a function of: (a),(e) workload size; (b),(f) average pattern size; (c),(g) pattern time window; (d),(h) pattern count-based window.

dataset were extremely similar to those obtained for time-based windows due to much less skew in event distribution over the input stream.

6.2.2 State-of-the-art Comparison. We repeated the experiments summarized in Figures 10 and 11 for the basic sharing and the basic reordering methods, as well as for two recent state-of-the-art MCEP methods [54, 64].

The basic sharing method (SH) refers to the maximal subexpression sharing technique mentioned in Section 3.2 and used in many previous studies (e.g., [7, 22, 24, 35]). The basic reordering method (RE) greedily rebuilds the event sequence by picking the event type maximizing the cost function at each step [40].

SPASS [54] selects the subpatterns to share according to a metric called 'redundancy ratio'. This metric represents the potential gain in sharing its computation. Each subexpression is assigned a score, and the winners are chosen by approximating the well-known minimal substring cover problem. MOTTO [64] utilizes a combination of techniques referred to as MST (merge sharing technique), DST (decomposition sharing technique), and OTT (operator transformation technique). The system solves the directed Steiner minimum tree problem to select the best global plan produced using the above techniques.

Figure 12 presents the results. The redundancy ratio method and the merge-decomposition technique are marked as SH-RR and SH-MDT respectively. While both SH-RR and SH-MDT scale well with growing workload size (Figures 12(a) and 12(e)) and average pattern length (Figures 12(f) and 12(f)), our optimizer achieves the best overall speedup, in some cases up to three times better than that of the runner-up solution. This result follows from utilizing the reordering opportunities, which were shown to drastically boost CEP evaluation [42]. On the other hand, our approach also attempts to exploit sharing opportunities when possible, which allows it to outperform the pure reordering algorithm (RE) for large pattern sizes. The gaps were closer for time window evaluation (Figures 12(c) and 12(g)), with SA-EDGE still achieving an advantage of at least 25% over the second-best method. The results for count-based windows (Figures 12(d) and 12(h)) strictly follow the trends described for Figure 11.

6.2.3 Adaptive System Behavior. We evaluated the performance of our system in the presence of a dynamically changing input stream. For this experiment alone, semi-synthetic input was used. We implemented a component that accepts a parameter x and randomly and independently transforms every x incoming events before they are received by the evaluation mechanism. A transformation is performed by

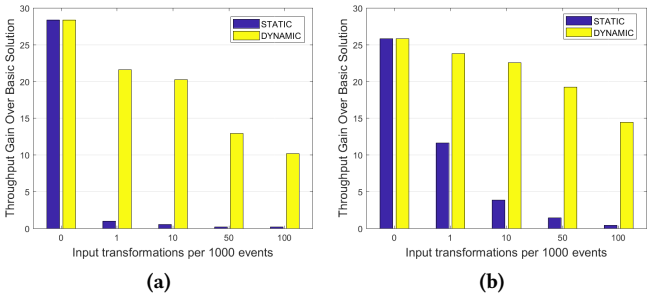

Figure 13: Throughput gain as a function of the number of arrival rate reshufflings per 1000 incoming events: (a) stock dataset; (b) traffic dataset.

randomly picking y event types, creating their random permutation P and then replacing the type attribute of every affected event with the one following its value in P. This modification allows us to simulate rapid and drastic changes in the arrival rates of all types of events.

We repeated the experiment for $y = 5$ and x ranging between 10 and 1000 on the static and the dynamic version of our framework. In the static case, an evaluation plan was created on startup and used exclusively regardless of input changes. The dynamic version utilized the adaptive approach introduced in [49], restarting the plan calculation process when a drastic change in the statistics is detected. The results are depicted in Figure 13. Unsurprisingly, the initially generated plan fails to perform adequately when the input characteristics overcome on-the-fly changes. While extremely frequent input changes clearly reduce system performance, the adaptive method still leads to at least 10 times higher throughput.

7 RELATED WORK

Complex event processing systems. Scalable solutions for real-time complex event detection have been the focus of much research in recent years [20, 21, 27]. Following the success of earlier data stream management systems [4, 10, 16, 17], a plethora of general purpose CEP frameworks were developed, including SASE/SASE+ [6, 62], CEDR [13], T-Rex [19], Amit [5] and ZStream [49]. Multiple CEP libraries are available, such as Esper [2], Siddhi [60], and Cayuga [22]. Moreover, large-scale CEP engines are on the rise, such as System S [9], TIBCO [15], WSO2 CEP [53], and CHAOS [29].

Single-pattern optimization. CEP systems implement a broad variety of optimization techniques aimed at minimizing processing time and resource consumption of a single pattern [19, 22, 25, 33, 50, 62, 63]. Multiple works focused exclusively on pattern rewriting. In [52], a rewriting framework based on unifying and splitting patterns is presented. ZStream [49] presents a dynamic programming algorithm for tree-based plan generation, utilizing a cost model similar to ours. NextCEP [57] assigns a cost to every candidate plan and utilizes a search algorithm to select the lowest cost evaluation scheme. In [7], events are processed in an ascending order of their arrival rates to optimize distributed CEP.

Multi-query optimization techniques. Multi-query optimization (MQO) is a well-known problem in database query processing [59]. Various methods have been proposed for sharing common subexpressions between queries in a workload. Notable examples include Volcano [56], MQJoin [47], and Monet [48]. Multi-query sharing techniques were incorporated in large-scale engines, such as SPARQL [43] and Microsoft SQL Server [65]. SWO [28] is the closest in spirit to our work. Rather than searching for common subexpressions, the global plan is calculated using an optimization algorithm that employs the branch-and-bound method.

Query sharing mechanisms were also developed for stream processing [11, 26, 30, 35, 38]. NiagaraCQ [17] is a large-scale system for processing multiple continuous queries over streams. Its supported features include dynamic workload modification. TelegraphCQ [16] is remarkable for introducing CACQ [46], a technique based on per-tuple dynamic routing [12] for inter-query sharing. MQO solutions were also proposed for processing XML streams [24, 34].

Pattern sharing techniques for complex event processing are being actively researched. Numerous advanced methods have been proposed for intra-pattern (sharing of subexpressions inside a nested pattern) [44, 55] and inter-pattern scenarios (sharing between different patterns) [7, 22, 45, 54, 64]. Some solutions consider aggregations [51].

To the best of our knowledge, PB-CED [7] and MOTTO [64] are the only CEP systems to consider a combination of sharing and pattern rewriting techniques. However, the solution provided by [7] only considers a single shared subpattern. In [64], arbitrary subset sharing is achieved by transforming a sequence pattern to a conjunction (rather than by reordering methods), which is known to severely diminish the performance of NFA-based event detection [41, 49].

8 CONCLUSIONS AND FUTURE WORK

In this paper, we studied the problem of optimizing multi-pattern CEP performance using a combination of sharing and pattern reordering techniques. We formally defined the respective optimization problem and presented an optimization framework for solving this computationally hard problem under tight real-time conditions. Our experimental evaluation demonstrated a significant performance boost as compared to state-of-the-art MCEP techniques. Our future research will further address additional challenges of MCEP, such as SLA support and dynamic workload modification.

REFERENCES

[1] http://www.eoddata.com.

[2] http://www.espertech.com.

[3] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[5] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[6] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[7] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.

[8] M. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *Proceedings of ISWC 2015 - 14th International Semantic Web Conference*, pages 374–389, Bethlehem, PA, USA, 2015. W3C.

[9] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, pages 27–37, New York, NY, USA, 2006. ACM.

[10] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford Data Stream Management System*, pages 317–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[11] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.

[12] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[14] M. Blount, M. Ebling, J. Eklund, A. James, C. Mcgregor, N. Percival, K. Smith, and D. Sow. Real-time analysis for intensive care: Development and deployment of the Artemis analytic system. 29:110–8, 05 2010.

[15] P. Brown. *Architecting Complex-Event Processing Solutions with TIBCO*. Addison-Wesley Professional, 1st edition, 2013.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.

[18] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proceedings of the 5th International Conference on Database Theory*, ICDT '95, pages 54–67, London, UK, 1995. Springer-Verlag.

[19] G. Cugola and A. Margara. Complex event processing with T-REX. *J. Syst. Softw.*, 85(8):1709–1728, 2012.

[20] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[21] M. Dayarathna and S. Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):33:1–33:36, February 2018.

[22] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, pages 627–644. Springer-Verlag.

[23] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[24] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, December 2003.

[25] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W. P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. *IEEE 24th International Conference on Data Engineering (ICDE)*, 0:676–685, 2008.

[26] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Data Stream Management*, Data-Centric Systems and Applications, pages 241–261. Springer, 2016.

[27] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, and M. Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217 – 236, 2017.

[28] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, 7(6):429–440, 2014.

[29] C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *2009 IEEE Conference on Commerce and Enterprise Computing*, pages 33–40, July 2009.

[30] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 297–308. VLDB Endowment, 2003.

[31] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of 37th Ann. IEEE Symp. on Foundations of Computer Science*, pages 627–636. IEEE Computer Society, 1996.

[32] M. Hill, M. Campbell, Y. C. Chang, and V. Iyengar. Event detection in sensor networks for modern oil fields. In *DEBS*, volume 332 of *ACM International Conference Proceeding Series*, pages 95–102. ACM, 2008.

[33] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[34] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 761–772, New York, NY, USA, 2007. ACM.

[35] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 120–131, New York, NY, USA, 2009. ACM.

[36] H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[37] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[38] P. Jovanovic, O. Romero, A. Simitsis, and A. Abello. Incremental consolidation of data-intensive multi-flows. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1203–1216, May 2016.

[39] I. Kolchinsky and A. Schuster. Efficient adaptive detection of complex event patterns. *PVLDB*, 11(11):1346–1359, 2018.

[40] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. *PVLDB*, 11(11):1332–1345, 2018.

[41] I. Kolchinsky, A. Schuster, and D. Keren. Efficient detection of complex event patterns using lazy chain automata. *CoRR*, abs/1612.05110, 2016.

[42] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45. ACM, 2015.

[43] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 666–677, Washington, DC, USA, 2012. IEEE Computer Society.

[44] M. Liu, E. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested CEP query processing over event streams. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*, pages 123–134.

[45] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 889–900, New York, NY, USA, 2011. ACM.

[46] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60, New York, NY, USA, 2002. ACM.

[47] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *Proc. VLDB Endow.*, 9(6):480–491, January 2016.

[48] S. Manegold, A. Pellenkoft, and M. Kersten. A multi-query optimizer for monet. In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, BNCOD 17, pages 36–50, London, UK, UK, 2000. Springer-Verlag.

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

[50] O. Poppe, C. Lei, S. Ahmed, and E. Rundensteiner. Complete event trend detection in high-rate event streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 109–124, New York, NY, USA, 2017. ACM.

[51] Y. Qi, L. Cao, M. Ray, and E. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 229–240, New York, NY, USA, 2014. ACM.

[52] E. Rabinovich, O. Etzion, and A. Gal. Pattern rewriting framework for event processing optimization. In *Proceedings of the 5th ACM International Conference on Distributed Event-based Systems*, pages 101–112. ACM, 2011.

[53] S. Ravindra and M. Dayarathna. Distributed scaling of wso2 complex event processor. 2015. https://wso2.com/library/articles/2015/12/article-distributed-scaling-of-wso2-complex-event-processor/.

[54] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *Proceedings of the 2016 International Conference on Management of Data*, pages 495–510, New York, NY, USA, 2016. ACM.

[55] M. Ray, E. Rundensteiner, M. Liu, C. Gupta, S. Wang, and I. Ari. High-performance complex event processing using continuous sliding views. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 525–536, New York, NY, USA, 2013. ACM.

[56] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 249–260, New York, NY, USA, 2000. ACM.

[57] N. P. Schultz-Møller, M. M., and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*. ACM, 2009.

[58] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Conference*, pages 23–34, 1979.

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

[60] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[61] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. *SIGMOD Rec.*, 18(2):367–376, 1989.

[62] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418. ACM, 2006.

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

[64] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. Multi-query optimization for complex event processing in SAP ESP. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1213–1224, 2017.

[65] J. Zhou, P. Larson, J. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 533–544, New York, NY, USA, 2007. ACM.

[66] Q. Zhou, Y. Simmhan, and V. K. Prasanna. Incorporating semantic knowledge into dynamic data processing for smart power grids. In *International Semantic Web Conference (2)*, volume 7650 of *Lecture Notes in Computer Science*, pages 257–273. Springer, 2012.

A EFFICIENT IMPLEMENTATION OF THE MULTI-PATTERN GRAPH

As presented in Section 4.1, the multi-pattern graph for the workload $WL = \{P_1, \cdots, P_n\}$ is defined as $MPG = (V, E)$, where $E = \left\{e_i = \left(v_i, v_j, \Gamma_{ij}\right) | v_i, v_j \in V, \Gamma_{ij} \neq \varnothing\right\}$ and $V = \{v_i | P_i \in WL\}$.

This formulation introduces potential performance issues. First, explicitly storing the set of common subpatterns Γ_{ij} requires $O\left(2^s\right)$ memory, where s is the size of the maximal common subpattern. This can be solved by only storing the MP_{ij} instead, as the rest of the common subpatterns can be inferred from it. Second, when m patterns share the same subpattern, the MPG will contain $\binom{m}{2}$ edges representing the same subpattern set. Consequently, directly instantiating the MPG in memory would be extremely inefficient.

We address this shortcoming by compact graph representation. Rather than explicitly store the vertices and the edges, for every distinct maximal common subpattern MP of some set of patterns Γ, we keep Γ in a hash table with MP as a key. In addition, a second hash table maps a single pattern P to a list of maximal common subpatterns with its peers in MPG. This data structure still contains all the

necessary information, additionally providing near constant cost of retrieval and worst case linear cost of addition and deletion of patterns. The space occupied by both hash tables is $O(n \cdot \gamma)$, where γ is the total number of distinct maximal common subpatterns in the workload. While the value of γ can reach n^2 in the worst case (and even exceed it in some cases that we describe shortly), the way in which the hash tables are constructed makes it extremely unlikely for the space complexity to surpass $O(n^2)$.

Another potential performance bottleneck associated with the MPG is the resource-consuming operation of calculating the maximal common subpatterns for all pairs of patterns. We will utilize the following simple and efficient implementation. Given $P_i = (\mathcal{E}_i, S_i, C_i, W_i)$ and $P_j = (\mathcal{E}_j, S_j, C_j, W_j)$, first a simple set intersection $\mathcal{E}_{ij}$ of $\mathcal{E}_i$ and $\mathcal{E}_j$ is calculated. Then, we project the conditions in C_i and C_j on S_{ij} and compare the resulting condition sets. If the sets are not equal, we calculate their intersection and reduce $\mathcal{E}_{ij}$ accordingly. The same procedure is then performed for S_i and S_j. Overall, the worst-case complexity of this operation is $O(max(|\mathcal{E}_i|, |\mathcal{E}_j|) + max(|C_i|, |C_j|))$.

For example, consider the above calculation for the workload consisting of $P_1 : AND(A, B, C); C_1 = \{A.x < 10\}$ and $P_2 : AND(A, D, C); C_2 = \{A.x \geq 10\}$. The intersection of event types in this case is $\mathcal{E}_{12} = \{A, B\}$. In addition, due to the conflicting conditions on A, the maximal common subpattern is reduced to $AND(B)$.

Note that multiple maximal common subpatterns may exist. For example, both $SEQ(A, B)$ and $SEQ(A, C)$ are the maximal intersections of the sequences $SEQ(A, B, C)$ and $SEQ(A, C, B)$. In this case, the MPG will store a list of maximal common subpatterns.

The worst-case complexity of computing all maximal common subpatterns is then $O(n^2 \cdot (s_{max} + c_{max}))$, where s_{max} and c_{max} denote the maximum sizes of a pattern in terms of events and conditions, respectively.

B LOCAL SEARCH META-HEURISTICS

This appendix provides a brief description of the most widely used local search meta-heuristics, *simulated annealing* and *Tabu search*. The reader is referred to [3, 36] for more details.

Simulated annealing extends the functionality of iterative improvement by also allowing limited non-improving moves. A threshold c_k is defined for each step. When a better neighbor solution is selected, it is chosen to replace the current solution, in a manner similar to the iterative improvement algorithm. If the neighbor solution is more expensive, it is accepted with probability $exp\left(-\frac{\triangle f}{c_k}\right)$, where $\triangle f$ is the difference between the costs of the old and the new solutions. The thresholds are chosen such that $c_k = \alpha \cdot c_{k-1}, \alpha < 1$. The algorithm starts with a sufficiently large c_0 and terminates when a predefined small value $c_{\bar{k}}$ is reached. Before the start of the actual search, c_0 is set to the largest difference observed during evaluation of I neighbors of s_{init}. In our experiments in Section 6, we used $\alpha = 0.99$ and $I = 10^3$ neighbors for setting the initial threshold.

Tabu search explores L random neighbors during each step and moves to the cheapest of them. Visiting the same state twice is prohibited. To enforce that, previously visited solutions are stored in a dedicated *tabu list*. The tabu list has a finite capacity C: when the number of stored solutions reaches C, oldest stored solutions are removed. The best solution $s*$ observed during the run of the algorithm is returned. We used a memory list of capacity $C = 10^4$ and $L = 100$ during our experimental evaluation.

Both algorithms stop after reaching a predefined number of steps since the last improvement to $s*$ or when the time expires. To study the tradeoff between evaluation time and solution quality, we only implemented the timestamp-based stop condition.

C FORMAL DEFINITION OF M-MCEP

In this appendix, we formally define the cost function and the optimization problem of multitree-based MCEP.

We start with extending the cost function. Let T_i denote a local tree-based evaluation plan for a pattern P_i. We borrow the cost function definition for tree-based plans from [40]. For a plan T_i, we define $Cost_{tree}(T) = \sum_{N \in nodes(T)} C(N)$, where

$$C(N) = \begin{cases} W_i \cdot r_j & N \text{ is a leaf representing } E_j \\ C(L) \cdot C(R) \cdot sel_{L,R} & N \text{ is an internal node with} \\ & \text{child nodes } L \text{ and } R. \end{cases}$$

Here, $sel_{L,R}$ denotes the total selectivity of all conditions defined between the event types in L and R.

The extension of $Cost_{tree}$ for multitrees will be defined by counting the individual costs of all nodes in a multitree:

$$Cost^{multi}_{tree}(MPM) = \sum_{N \in nodes(MPM)} C(N).$$

Given a tree-based plan T and a multi-pattern multitree MPM, we will say that $T \in MPM$ if and only if MPM contains a subtree identical to T. We will denote a subtree of the MPM corresponding to a pattern p_i as $\mathcal{T}_i$. In addition, we will denote by $TREE_P$ the set of all tree-based plans of a pattern P. The extended optimization problem will be subsequently defined as follows.

Multitree-based multi-pattern CEP optimization problem (M-MCEP). Given a workload WL of n patterns and a statistics collection $Stat$, find a multi-pattern multitree MPM minimizing the value of $Cost^{multi}_{tree}(MPM, WL, Stat)$ subject to $\forall P_i, 1 \leq i \leq n : \exists T \in TREE_P$ s.t. $T \in MPM$.

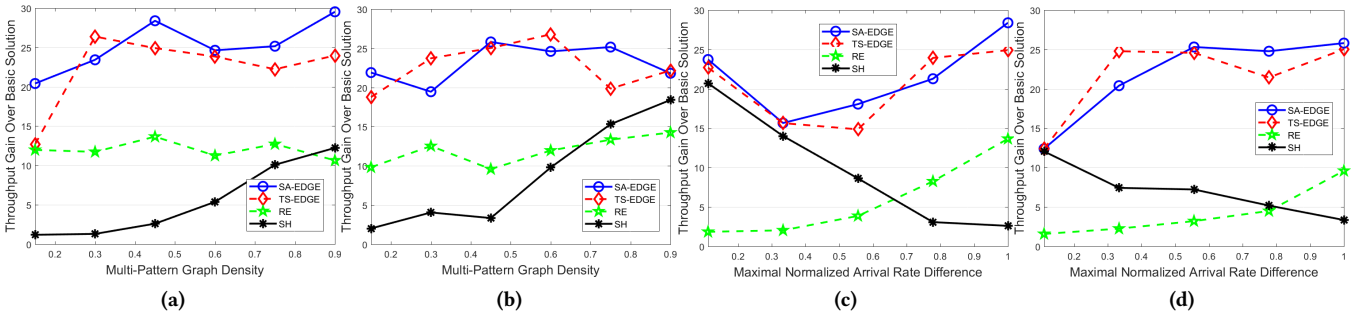

Figure 14: Throughput gain of the local search algorithms as a function of workload statistical properties: (a) stock dataset, sharing sensitivity; (b) traffic dataset, sharing sensitivity; (c) stock dataset, reordering sensitivity; (d) traffic dataset, reordering sensitivity.

Since T-MCEP can be viewed as a particular case of M-MCEP (restricted to left-deep trees as local plans), the complexity results obtained for T-MCEP in Section 3.4 hold for M-MCEP by generalization.

To justify the use of $\mathcal{N}_{edge}$ and $\mathcal{N}^k_{vertex}$ for MPM-based solution space, we utilize an observation similar to the one presented in Theorem 4.1.

THEOREM C.1. *Let MPM_{opt} be the optimal multi-pattern multitree for some workload W. For each tree $\mathcal{T}_i$ in MPM_{opt} corresponding to the pattern P_i, let $\mathcal{S}_i$ denote the set of subtrees that are shared with other patterns in MPM_{opt}. Then, $\mathcal{T}_i$ is the most efficient local tree-based plan for P_i out of those containing all the subtrees in $\mathcal{S}_i$.*

D ADDITIONAL EXPERIMENTS

In this appendix, we experimentally study the influence of the workload statistical characteristics on the performance of our optimizer. Only the best performing (according to the results presented in Section 6.2) combinations SA-EDGE and TS-EDGE were evaluated.

We control the statistical characteristics of workload generation using a pair of configurable parameters, *multi-pattern graph density* and *normalized arrival rate difference*. The multi-pattern graph density is defined as an average relative number of neighbors of a given pattern in a MPG. For example, in a workload of 100 patterns with MPG density equal to 0.5, each pattern will have 50 neighbors on average. This parameter is used to control the *sharing sensitivity* of a workload.

The arrival rate difference, defined as the maximal difference in rates of two event types within a single pattern, allows us to manipulate the *reordering sensitivity* of a workload. For example, for an unconditional conjunction of 5 event types arriving at an identical rate, each of the possible 5! evaluation orders will have the same cost. However, if one of the types appears 100 times more frequently than the rest, the gain obtained by postponing the costly event type to the last state is considerably high. Patterns with varying degrees of reordering sensitivity are produced by limiting the selection of the event types for a pattern accordingly. The values of this parameter were normalized with respect to the maximal observed difference of 45.

Figure 14 depicts the achieved throughput gain as a function of the sharing sensitivity (Figures 14(a) and 14(b)) and the reordering sensitivity (Figures 14(c) and 14(d)) of the workload. The plots also show the performance of the basic reordering (RE) and the basic sharing (SH) methods discussed in Section 6.2.

The high gains of the local search methods do not exhibit dominant dependencies on either of the two parameters. While larger graph densities and rate difference limits introduce more sharing and reordering opportunities, they also increase the search space size and the number of potential local minima. Nevertheless, our approach consistently outperforms the better of SH and RE for every attempted experimental configuration. At the extremes, local search tends to resort to an almost pure sharing plan for low arrival rate differences (since virtually no improvement can be achieved by reordering), whereas for sparse multi-pattern graphs the solution assigning the best local plan to all patterns is often preferred.

The basic reordering method becomes more efficient with increasing differences in arrival rate and is almost unaffected by the changes in graph density. The performance of the basic sharing method increases monotonically with graph density. It also decreases with the rate difference due to the smaller number of participating event types in more restricted workloads. Given a pair of workloads of the same size containing patterns of the same length, the workload with fewer event types will have more events of the same type on average, and is expected to offer more sharing opportunities.