Join Query Optimization Techniques for Complex Event Processing Applications

Ilya Kolchinsky
Technion, Israel Institute of Technology
Haifa 32000 Israel
ikolchin@cs.technion.ac.il

Assaf Schuster
Technion, Israel Institute of Technology
Haifa 32000 Israel
assaf@cs.technion.ac.il

ABSTRACT

Complex event processing (CEP) is a prominent technology used in many modern applications for monitoring and tracking events of interest in massive data streams. CEP engines inspect real-time information flows and attempt to detect combinations of occurrences matching predefined patterns. This is done by combining basic data items, also called "primitive events", according to a pattern detection plan, in a manner similar to the execution of multi-join queries in traditional data management systems. Despite this similarity, little work has been done on utilizing existing join optimization methods to improve the performance of CEP-based systems.

In this paper, we provide the first theoretical and experimental study of the relationship between these two research areas. We formally prove that the CEP Plan Generation problem is equivalent to the Join Query Plan Generation problem for a restricted class of patterns and can be reduced to it for a considerably wider range of classes. This result implies the NP-completeness of the CEP Plan Generation problem. We further show how join query optimization techniques developed over the last decades can be adapted and utilized to provide practically efficient solutions for complex event detection. Our experiments demonstrate the superiority of these techniques over existing strategies for CEP optimization in terms of throughput, latency, and memory consumption.

PVLDB Reference Format:
Ilya Kolchinsky and Assaf Schuster. Join Query Optimization Techniques for Complex Event Processing Applications. *PVLDB*, 11 (11): 1332-1345, 2018.
DOI: https://doi.org/10.14778/3236187.3236189

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07... $ 10.00.
DOI: https://doi.org/10.14778/3236187.3236189

Keywords

Stream Processing, Complex Event Processing, Lazy Evaluation, Query Optimization

1. INTRODUCTION

Complex event processing has become increasingly important for applications in which arbitrarily complex patterns must be efficiently detected over high-speed streams of events. Online finance, security monitoring, and fraud detection are among the many examples. Pattern detection generally consists of collecting primitive events and combining them into potential (partial) matches using some type of detection model. As more events are added to a partial match, a full pattern match is eventually formed and reported. Popular CEP mechanisms include nondeterministic finite automata (NFAs) [5, 18, 51], finite state machines [6, 45], trees [36], and event processing networks [21, 42].

A CEP engine creates an internal representation for each pattern P to be monitored. This representation is based on a model used for detection (e.g., an automaton or a tree) and reflects the structure of P. In some systems [5, 51], the translation from a pattern specification to a corresponding representation is a one-to-one mapping. Other frameworks [6, 30, 36, 42, 45] introduce the notion of a *cost-based evaluation plan*, where multiple representations of P are possible, and one is chosen according to the user's preference or some predefined cost metric.

We will illustrate the above using the following example. Assume that we are receiving periodical readings from four traffic cameras A, B, C and D. We are required to recognize a sequence of appearances of a particular vehicle on all four cameras in order of their position on a road, e.g., $A \rightarrow B \rightarrow C \rightarrow D$. Assume also that, due to a malfunction in camera D, it only transmits one frame for each 10 frames sent by the other cameras.

Figure 1(a) displays a nondeterministic finite automaton (NFA) for detecting this pattern, as described in [51]. A state is defined for each prefix of a valid match. During evaluation, a combination of camera readings matching each prefix will be represented by a unique instance of the NFA in the corresponding state. Transitions between states are triggered nondeterministically by the arrival of an event satisfying the constraints defined by the pattern. A new NFA instance is created upon each transition.

The structure of the above automaton is uniquely dictated by the order of events in the given sequence. However, due to the low transmission rate of D, it would be beneficial to wait for its signal before examining the local history for previous

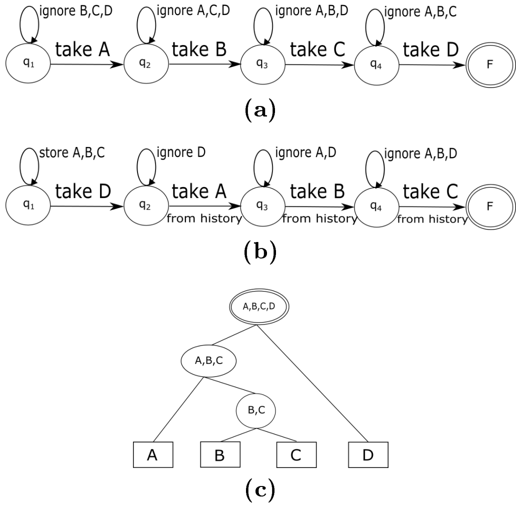

Figure 1: Evaluation structures for a CEP pattern *SEQ(A,B,C,D)*: (a) NFA with no reordering; (b) NFA with reordering; (c) evaluation tree.

readings of A, B and C that match the constraints. This way, fewer prefixes would be created. Figure 1(b) demonstrates an out-of-order NFA for the rewritten pattern (defined as "Lazy NFA" in [30]). It starts by monitoring the rarest event type D and storing the other events in the dedicated buffer. As a reading from camera D arrives, the buffer is inspected for events from A, B and C preceding the one received from D and located in the same time window. This plan is more efficient than the one implicitly used by the first NFA in terms of the number of partial matches created during evaluation. Moreover, unless more constraints on the events are defined, it is the best out of all (4!) possible plans, that is, mutual orders of A, B, C and D.

Not all CEP mechanisms represent a plan as an evaluation order. Figure 1(c) depicts a tree-based evaluation mechanism [36] for detecting the above pattern. Events are accepted at the corresponding leaves of the tree and passed towards the root where full matches are reported. This model requires an evaluation plan to be supplied, because, for a pattern of size n, there are at least $n! \cdot C_{n-1} = \frac{(2n-2)!}{(n-1)!}$ possible trees (where C_n is the nth Catalan number) [37].

In many scenarios, we will prefer the evaluation mechanisms supporting cost-based plan generation over those mechanisms allowing for only one such plan to be defined. This way, we can drastically boost system performance subject to selected metrics by picking more efficient plans. However, as the space of potential plans is at least exponential in pattern size, finding an optimal plan is not a trivial task.

Numerous authors have identified and targeted this issue. Some of the proposed solutions are based on rewriting the original pattern according to a set of predefined rules to maximize the efficiency of its detection [42, 45]. Other approaches discuss various strategies and algorithms for generating an evaluation plan that maximizes the performance for a given pattern according to some cost function [6, 30, 36]. While the above approaches demonstrate promising results, this research field remains largely unexplored, and the space of the potential optimization techniques is still far from being exhausted.

The problem described above closely resembles the problem of estimating execution plans for large join queries. As opposed to CEP plan generation, this is a well-known, established, and extensively targeted research topic. A plethora of methods and approaches producing close-to-optimal results were published during the last few decades. These methods range from simple greedy heuristics, to exhaustive dynamic programming techniques, to randomized and genetic algorithms [32, 33, 38, 46, 47, 48].

Both problems look for a way to efficiently combine multiple data items such that some cost function is minimized. Also, both produce solutions possessing similar structures. If we reexamine Figure 1, we can see that evaluation plans for NFAs (1(b)) and trees (1(c)) closely resemble left-deep tree plans and bushy tree plans [26] respectively. An interesting question is whether join-related techniques can be used to create better CEP plans using a proper reduction.

In this work, we attempt to close the gap between the two areas of research. We study the relationship between CEP Plan Generation (CPG) and Join Query Plan Generation (JQPG) problems and show that any instance of CPG can be transformed into an instance of JQPG. Consequently, any existing method for JQPG can be made applicable to CPG. Our contributions can be summarized as follows:

- We formally prove the equivalence of JQPG and CPG for a large subset of CEP patterns, the conjunctive patterns. The proof addresses the two major classes of evaluation plans, the order-based plans and the tree-based plans (Section 3).
- We extend the above result by showing how other pattern types can be converted to conjunctive patterns, thus proving that any instance of CPG can be reduced to an instance of JQPG (Section 4).
- The deployment of a JQPG method to CPG is not trivial, as multiple CEP-specific issues need to be addressed, such as detection latency constraints, event consumption policies, and adaptivity considerations. We present and discuss the steps essential for successful adaptation of JQPG techniques to the CEP domain (Section 5).
- We validate our theoretical analysis in an extensive experimental study. Several well-known JQPG methods, such as Iterative Improvement [48] and Dynamic Programming [46], were applied on a real-world event dataset and compared to the existing state-of-the-art CPG mechanisms. The results demonstrate the superiority of the adapted JQPG techniques (Section 6).

2. BACKGROUND AND TERMINOLOGY

In this section, we introduce the notations used throughout this paper, provide the necessary background on complex event processing and describe the two problems whose relationship will be closely studied in the next sections.

2.1 CEP Patterns

The patterns recognized by CEP systems are normally formed using declarative specification languages [14, 18, 51]. A pattern is defined by a combination of primitive events, operators, a set of predicates to be satisfied by the participating events, and a time window. Each event is represented by a type and a set of attributes, including the occurrence timestamp. We assume that each primitive event has a well-defined type, i.e., the event either contains the type as an attribute or it can be easily inferred from other attributes

using negligible system resources. The operators describe the relations between different events comprising a pattern match. The predicates, usually organized in a Boolean formula, specify the constraints on the attribute values of the events. As an example, consider the following pattern specification syntax, taken from SASE [51]:

$$\texttt{PATTERN}\; op\,(T_1\, e_1, T_2\, e_2, \cdots, T_n\, e_n)$$
$$\texttt{WHERE}\,(c_{1,1} \wedge c_{1,2} \wedge \cdots \wedge c_{n,n-1} \wedge c_{n,n})$$
$$\texttt{WITHIN W.}$$

Here, the PATTERN clause specifies the events $e_1, \cdots, e_n$ we would like to detect and the operator *op* to combine them (see below). The WHERE clause defines a Boolean CNF formula of inter-event constraints, where $c_{i,j}; 1 \leq i, j \leq n$ stands for the mutual condition between attributes of e_i and e_j. $c_{i,i}$ declares filter conditions on e_i. Any of $c_{i,j}$ can be empty. For the rest of our paper, we assume that all conditions between events are at most pairwise. This assumption is for presentational purposes only, as our results can be easily generalized to arbitrary predicates. The WITHIN clause sets the time window W, which is the maximal allowed difference between the timestamps of any pair of events in a match.

In this paper, we will consider the most commonly used operators, namely AND, SEQ, and OR. The AND operator requires the occurrence of all events specified in the pattern within the time window. The SEQ operator extends this definition by also expecting 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 operators of particular importance are the negation operator (NOT) and the Kleene closure operator (KL). They can only be applied on a single event and are used in combination with other operators. $NOT\,(e_i)$ requires the absence of the event e_i from the stream (or from a specific position in the pattern in the case of the SEQ operator), whereas $KL\,(e_i)$ accepts one or more instances of e_i. In the remainder of this paper, we will refer to NOT and KL as *unary operators*, while AND, SEQ and OR will be called *n-ary operators*.

The PATTERN clause may include an unlimited number of n-ary and unary operators. We will refer to patterns containing a single n-ary operator, and at most a single unary operator per primitive event, as *simple patterns*. On the contrary, *nested patterns* are allowed to contain multiple n-ary operators (e.g., a disjunction of conjunctions and/or sequences will be considered a nested pattern). Nested patterns present an additional level of complexity and require advanced techniques (e.g., as described in [34]).

We will further divide simple patterns into subclasses. A simple pattern whose n-ary operator is an AND operator will be denoted as a *conjunctive pattern*. Similarly, *sequence pattern* and *disjunctive pattern* will stand for patterns with SEQ and OR operators, respectively. A simple pattern containing no unary operators will be called a *pure pattern*.

The "four cameras pattern" described in Section 1 illustrates the above. This is a pure sequence pattern, written in SASE as follows:

```
PATTERN SEQ(A a,B b,C c,D d)
WHERE(a.vehicleID=b.vehicleID=
     =c.vehicleID=d.vehicleID)
WITHIN W.
```

2.2 Order-based Evaluation Mechanisms

Order-based evaluation mechanisms play an important role in CEP engines based on state machines. One of the most commonly used models following this principle is the NFA (nondeterministic finite automaton) [5, 18, 51]. An NFA consists of a set of states and conditional transitions between them. Each state corresponds to a prefix of a full pattern match. Transitions are triggered by the arrival of the primitive events, which are then added to partial matches. Conditions between events are verified during the transitions. Figure 1(a) depicts an example of an NFA constructed for the "four cameras" sequence pattern. While in theory NFAs may possess an arbitrary topology, non-nested patterns are normally detected by a chain-like structure.

The basic NFA model does not include any notion of altering the "natural" evaluation order or any other optimization based on pattern rewriting. Multiple works have presented methods for constructing NFAs with out-of-order processing support. W.l.o.g., we will use the Lazy NFA mechanism, a chain-structured NFA introduced in [29, 30].

Given a pattern of n events and a user-specified order O on the event types appearing in the pattern, a chain of $n+1$ states is constructed, with each state k corresponding to a match prefix of size $k-1$. The order of the states matches O. If a type appears more than once in a pattern, it will also appear multiple times in O. The $(n+1)^{th}$ state in the chain is the accepting state. To achieve out-of-order evaluation, incoming events are stored locally. A buffered event is retrieved and processed when its corresponding state in the chain is reached. Figure 1(b) presents an example of this construction for $O = (D, A, B, C)$.

This construction method allows us to apply all possible $(n!)$ orders without affecting the detection correctness.

2.3 Tree-based Evaluation Mechanisms

An alternative to NFA, the tree-based evaluation mechanism [36] specifies which subsets of full pattern matches are to be tracked by defining tree-like structures. For each event participating in a pattern, a designated leaf is created. During evaluation, events are routed to their corresponding leaves and are buffered there. The non-leaf nodes accumulate the partial matches. The computation at each non-leaf node proceeds only when all of its children are available (i.e., all events have arrived or partial matches have been calculated). Matches formed at the tree root are reported to the end users. An example is shown in Figure 1(c).

ZStream assumes a batch-iterator setting [36]. To perform our study under a unified framework, we modify this behavior to support arbitrary time windows. As described above with regard to NFAs, a separate tree instance will be created for each currently found partial match. As a new event arrives, an instance will be created containing this event. Every instance I corresponds to some subtree s of the tree plan, with the leaves of s holding the primitive events in I. Whenever a new instance I' is created, the system will attempt to combine it with previously created "siblings", that is, instances corresponding to the subtree sharing the parent node with s'. As a result, another new instance containing the unified subtree will be generated. This in turn will trigger the same process again, and it will proceed recursively until the root of the tree is reached or no siblings are found.

ZStream includes an algorithm for determining the optimal tree structure for a given pattern. This algorithm is

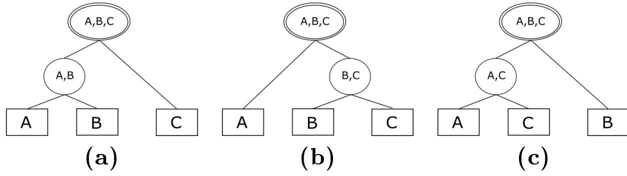

Figure 2: Evaluation trees for a pattern *SEQ(A,B,C)*: (a) a left-deep tree produced by ZStream; (b) a right-deep tree produced by ZStream; (c) an optimal evaluation tree, which cannot be produced by ZStream.

based on a cost model that takes into account the arrival rates of the primitive events and the selectivities of their predicates. However, since leaf reordering is not supported, a subset of potential plans is missed. We will illustrate this drawback using the following example:

```
PATTERN SEQ(A a,B b,C c)
WHERE(a.x=c.x) WITHIN W.
```

We assume that all events arrive at identical rates, and that the condition between A and C is very restrictive. Figures 2(a) and 2(b) present the only two possible plans according to the algorithm presented in [36]. However, due to the condition between A and C, the most efficient evaluation plan is the one displayed in Figure 2(c).

2.4 CEP Plan Generation

We will start with the definition of the CEP evaluation plan. The *evaluation plan* provides a scheme for the evaluation mechanism, according to which its internal pattern representation is created. Therefore, different evaluation plans are required for different CEP frameworks. In this paper, we distinguish between two main types of plans, the *order-based plan* and the *tree-based plan*.

An order-based plan consists of a permutation of the primitive event types declared by the pattern. A CEP engine uses this plan to set the order in which events are processed. Order-based plans are applicable to mechanisms evaluating a pattern event-by-event, as described in Section 2.2.

A tree-based plan extends the above by providing a tree-like scheme for pattern evaluation. It specifies which subsets of valid matches are to be locally buffered and how to combine them into larger partial matches. Plans of this type are used by the evaluation mechanism presented in Section 2.3.

We can thus define two variations of the CEP Plan Generation problem, *order-based CPG* and *tree-based CPG*. In each variation, the goal is to determine an optimal evaluation plan P subject to some cost function $Cost(P)$. Different CEP systems define different metrics to measure their efficiency. In this paper we will consider a highly relevant performance optimization goal: reducing the number of active partial matches within the time window (denoted below simply as *number of partial matches*).

Regardless of the system-specific performance objectives, the implicit requirement to monitor all valid subsets of primitive events can become a major bottleneck. Because any partial match might form a full pattern match, their number is worst-case exponential in the number of events participating in a pattern. Further, as a newly arrived event needs to be checked against all (or most of) the currently stored partial matches, the processing time and resource consumption per event can become impractical for real-time applications. Other metrics, such as detection latency or network communication cost, may also be negatively affected. Thus, given the crucial role of the number of partial matches in all aspects of CEP, it was chosen as our primary cost function.

2.5 Join Query Plan Generation

Join Query Plan Generation is a well-known problem in query optimization [32, 46, 48]. We are given relations $R_1, \cdots, R_n$ and a query graph describing the conditions to be satisfied by the tuples in order to be included in the result. A condition between a pair of relations R_i, R_j has a known selectivity $f_{i,j} \in [0,1]$. The goal is to produce a join query plan minimizing a predefined cost function.

One popular choice for the cost function is the number of intermediate tuples produced during plan execution. For the rest of this paper, we will refer to it as the *intermediate results size*. In [13], the following expression is given to calculate this function for each two-way join of two input relations:

$$C(R_i, R_j) = |R_i| \cdot |R_j| \cdot f_{i,j},$$

where $|R_i|, |R_j|$ are the cardinalities of the joined relations. This formula is naturally extended to relations produced during join calculation:

$$C(S,T) = |S| \cdot |T| \cdot f_{S,T}.$$

Here, $S = R_{i_1} \bowtie \cdots \bowtie R_{i_s}; T = R_{j_1} \bowtie \cdots \bowtie R_{j_t}$ are the partial join results of some subsets of $R_1, \cdots, R_n$ and $f_{S,T} = (|S \bowtie T| \,/\, |S \times T|)$ is the product of selectivities of all predicates defined between the individual relations comprising S and T.

The two most popular classes of join query plans are the left-deep trees and the bushy trees [26]. A join tree of the former type processes the input relations one-by-one, adding a new relation to the current intermediate result during each step. Hence, for this class of techniques, a valid solution is a join order rather than a join plan. Approaches based on bushy trees pose no limitations on the plan topology, allowing it to contain arbitrary branches.

JQPG was shown by multiple authors to be NP-complete [13, 24], even when only left-deep trees are considered.

3. THE EQUIVALENCE OF CPG AND JQPG FOR PURE CONJUNCTIVE PATTERNS

This section presents the formal proof of equivalence between CPG and JQPG for pure conjunctive patterns. We show that, when the pattern to be monitored is a pure conjunctive pattern and the CPG cost function represents the number of partial matches, the two problems are equivalent.

3.1 Order-Based Evaluation

We will first focus on a CPG variation for order-based evaluation plans. In this section we will show that this problem is equivalent to JQPG restricted to left-deep trees. To that end, we will define the cost model functions for both problems and then present the equivalence theorem.

Our cost function $Cost_{ord}$ will reflect the number of partial matches coexisting in memory within the time window. The calculations will be based on the arrival rates of the events and the selectivities of the predicates.

Let $sel_{i,j}$ denote the selectivity of $c_{i,j}$, i.e., the probability of a partial match containing instances of events

of types T_i and T_j to pass the condition. Additionally, let $r_1, \cdots r_n$ denote the arrival rates of corresponding event types $T_1, \cdots T_n$. Then, the expected number of primitive events of type T_i arriving within the time window W is $W \cdot r_i$. Let $O = (T_{p_1}, T_{p_2}, \cdots T_{p_n}); p_i \in [1, n]$ denote an execution order. Then, during pattern evaluation according to O, the expected number of partial matches of length $k, 1 \le k \le n$ is given by:

$$PM(k) = W^k \cdot \prod_{i=1}^{k} r_{p_i} \cdot \prod_{i,j \le k; i \le j} sel_{p_i,p_j}.$$

The overall cost function we will attempt to minimize is thus the sum of partial matches of all sizes, as follows:

$$Cost_{ord}(O) = \sum_{k=1}^{n} \left(W^k \cdot \prod_{i=1}^{k} r_{p_i} \cdot \prod_{i,j \le k; i \le j} sel_{p_i,p_j} \right).$$

For the JQPG problem restricted to output left-deep trees only, we will use the two-way join cost function $C(S, T)$ defined in Section 2.5. Let L be a left-deep tree and let $\{i_1, i_2, \cdots, i_n\}$ be the order in which input relations are to be joined according to L. Let $P_k, 1 \le k < n$ denote the result of joining the first k tables by L (that is, $P_1 = R_{i_1}$, $P_2 = R_{i_1} \bowtie R_{i_2}$, etc.). In addition, let $C_1 = |R_{i_1}| \cdot f_{i_1,i_1}$ be the cost of the initial selection from R_{i_1}. Then, the cost of L will be defined according to a left-deep join (LDJ) cost function:

$$Cost_{LDJ}(L) = C_1 + \sum_{k=2}^{n} C(P_{k-1}, R_{i_k}).$$

We are now ready to formally prove the statement formulated in the beginning of the section.

THEOREM 1. *Given a pure conjunctive pattern P, the problem of finding an order-based evaluation plan for P minimizing $Cost_{ord}$ is equivalent to the Join Query Plan Generation problem for left-deep trees subject to $Cost_{LDJ}$.*

We will only show here the reduction from CPG to JQPG, which will be used in the later sections to apply join plan generation algorithms on CEP patterns. The opposite direction is symmetric and can be found in [27].

Given a pure conjunctive pattern P defined over event types $T_1, \cdots, T_n$ with predicates $c_{i,j} : 1 \le i, j \le n$, let $R_1, \cdots, R_n$ be a set of relations such that each R_i corresponds to an event type T_i. For each attribute of T_i, including the timestamp, a matching column will be defined in R_i. The cardinality of R_i will be set to $W \cdot r_i$, and, for each predicate $c_{i,j}$ with selectivity $sel_{i,j}$, an identical predicate will be formed between the relations R_i and R_j. We will define the query corresponding to P as follows:

```
SELECT * FROM R_1, ... R_n
WHERE (c_{1,1} AND ... AND c_{n,n}).
```

We will show that a solution to this instance of the JQPG problem is also a solution to the initial CPG problem. Recall that a left-deep JQPG solution L minimizes the function $Cost_{LDJ}$. By opening the recursion and substituting the parameters with those of the original problem, we get:

$$Cost_{LDJ}(L) = C_1 + \sum_{k=2}^{n} C(P_{k-1}, R_{i_k}) =$$

$$= |R_{i_1}| \cdot f_{i_1,i_1} + \sum_{k=2}^{n} \left(\prod_{j=1}^{k} |R_{i_j}| \cdot \prod_{j.l \le k; j \le l} f_{i_j,i_l} \right) =$$

$$= \sum_{k=1}^{n} \left(\prod_{j=1}^{k} (W \cdot r_{i_j}) \cdot \prod_{j.l \le k-1; j \le l} sel_{i_j,i_l} \right) = Cost_{ord}(O).$$

Consequently, the solution that minimizes $Cost_{LDJ}$ also minimizes $Cost_{ord}$, which completes the proof.■

In [13] the authors showed the problem of Join Query Plan Generation for left-deep trees to be NP-complete for the general case of arbitrary query graphs. From this result and from the equivalence stated by Theorem 1 (proven in full in [27]) we will deduce the following corollary.

COROLLARY 1. *The problem of finding an order-based evaluation plan for a general pure conjunctive complex event pattern that minimizes $Cost_{ord}$ is NP-complete.*

3.2 Tree-Based Evaluation

In this section, we will extend Theorem 1 to tree-based evaluation plans. This time we will consider the unrestricted JQPG problem, allowed to return bushy trees. Similarly to Section 3.1, we will start by defining the cost functions and then proceed to the proof of the extended theorem.

We will define the cost model for evaluation trees in a manner similar to Section 3.1. We will estimate the number of partial matches accumulated in each node of the evaluation tree and sum them up to produce the cost function.

For a leaf node l collecting events of type T_i, the expected number of partial matches is equal to the number of events of type T_i arriving inside a time window:

$$PM(l) = W \cdot r_i.$$

To obtain an estimate for an internal node in, we multiply the cost function values of its children by the total selectivity of the predicates verified by this node:

$$PM(in) = PM(in.left) \cdot PM(in.right) \cdot SEL_{LR}(in),$$

where SEL_{LR} is the selectivity of the predicates defined between event types accepted at the left and the right subtrees of node in, or, more formally:

$$SEL_{LR}(in) = \prod_{e_i \in in.ltree; e_j \in in.rtree} sel_{i,j}.$$

The total cost function on a tree T is thus defined as follows:

$$Cost_{tree}(T) = \sum_{N \in nodes(T)} PM(N).$$

For bushy trees, we will extend the cost function defined in Section 3.1. The cost of a tree node N will be defined as follows:

$$C(N) = \begin{cases} |R_i| & N \text{ is a leaf representing } R_i \\ |L| \cdot |R| \cdot f_{L,R} & N \text{ is an internal node representing} \\ & a\ sub-join\ L \bowtie R, \end{cases}$$

with the bushy join (BJ) cost function defined as follows:

$$Cost_{BJ}(T) = \sum_{N \in nodes(T)} C(N).$$

We will now extend Theorem 1 to tree-based plans.

THEOREM 2. *Given a pure conjunctive pattern P, the problem of finding a tree-based evaluation plan for P minimizing $Cost_{tree}$ is equivalent to the Join Query Plan Generation problem subject to $Cost_{BJ}$.*

To prove the theorem, we decompose each of the tree cost functions $Cost_{tree}, Cost_{BJ}$ into two components, separately calculating the cost of the leaves and the internal nodes:
$Cost^{l}_{tree}(T) = \sum_{N \in leaves(T)} PM(N)$
$Cost^{in}_{tree}(T) = \sum_{N \in in_nodes(T)} PM(N)$
$Cost^{l}_{BJ}(T) = \sum_{N \in leaves(T)} C(N)$
$Cost^{in}_{BJ}(T) = \sum_{N \in in_nodes(T)} C(N)$.
Obviously, the following equalities hold:

$$Cost_{tree}(T) = Cost^{l}_{tree}(T) + Cost^{in}_{tree}(T)$$
$$Cost_{BJ}(T) = Cost^{l}_{BJ}(T) + Cost^{in}_{BJ}(T).$$

Thus, it is sufficient to prove $Cost^{l}_{tree}(T) = Cost^{l}_{BJ}(T)$ and $Cost^{l}_{tree}(T) = Cost^{in}_{BJ}(T)$ for every T. From here it will follow that the solution minimizing $Cost_{tree}$ will also minimize $Cost_{BJ}$ and vice versa.

Applying either direction of the reduction from Theorem 1, we observe the following for the first pair of functions:

$$Cost^{l}_{tree}(T) = \sum_{N \in leaves(T)} PM(N) = \sum_{i=1}^{n} W \cdot r_i =$$
$$= \sum_{i=1}^{n} |R_i| = \sum_{N \in leaves(T)} C(N) = Cost^{l}_{BJ}(T).$$

For the second pair of functions, we will first expand the recursion of $Cost^{in}_{tree}$:

$$Cost^{in}_{tree}(T) = \sum_{N \in in_nodes(T)} PM(N) =$$
$$= \sum_{N \in in_nodes(T)} PM(N.left) \cdot PM(N.right) \cdot SEL_{LR}(N) =$$
$$= \sum_{N \in in_nodes(T)} \left(\prod_{m \in leaves(N)} W \cdot r_m \cdot \prod_{i,j \in leaves(N)} sel_{i,j} \right),$$

where $leaves(N)$ denotes the leaves of a tree rooted at N.
By similarly opening the recursion of $Cost^{in}_{JB}$ we obtain:

$$Cost^{in}_{BJ}(T) =$$
$$\sum_{N \in in_nodes(T)} \left(\prod_{m \in leaves(N)} |R_m| \cdot \prod_{i,j \in leaves(N)} f_{i,j} \right).$$

After substituting $r_m = \frac{|R_m|}{W}$ and $sel_{p_i,p_j} = f_{p_i,p_j}$, the two expressions are identical, which completes the proof.■

The CPG-JQPG reduction that we will use for tree-based evaluation is the one demonstrated in Theorem 1 for order-based evaluation.

By Theorem 2 and the generalization of the result in [13], we derive the following corollary.

COROLLARY 2. *The problem of finding a tree-based evaluation plan for a general pure conjunctive complex event pattern that minimizes $Cost_{tree}$ is NP-complete.*

3.3 Join Query Types

As Corollaries 1 and 2 imply, no efficient algorithm can be devised to optimally solve CPG for a general conjunctive pattern unless $P = NP$. However, better complexity results may be available under certain assumptions regarding the pattern structure. Numerous works considered the JQPG problem for restricted *query types*, that is, specific topologies of the query graph defining the inter-relation conditions. Examples of such topologies include clique, tree, and star.

It was shown in [24, 32] that an optimal plan can be computed in polynomial time for left-deep trees and queries forming an acyclic graph (i.e., tree queries), provided that the cost function has the ASI (adjacent sequence interchange) property [39]. The left-deep tree cost function $Cost_{LDJ}$ has this property [13], making the result applicable for our scenario. A polynomial algorithm without the ASI requirement was proposed for bushy tree plans for chain queries [40]. From Theorems 1 and 2 we can conclude that, for conjunctive patterns only, CPG∈P under the above constraints.

However, these results only hold when the plans produced by a query optimizer are not allowed to contain cross products [13, 40]. While this limitation is well-known in relational optimization [50], it is not employed by the existing CPG methods [6, 30, 36, 45]. Thus, even when an exact polynomial algorithm is applicable to CPG, it is inferior to native algorithms in terms of the considered search space and can only be viewed as a heuristic. In that sense, it is similar to the greedy and randomized approaches [47, 48].

Other optimizations utilizing the knowledge of the query type were proposed. For example, the optimal bushy plan was empirically shown to be identical to the optimal left-deep plan for star queries and, in many cases, for grid queries [47]. This observation allows us to utilize a cheaper left-deep algorithm for the above query types without compromising the quality of the resulting plan.

With the introduction of additional pattern types (Section 4) and event selection strategies (Section 5.2), new query graph topologies might be identified and type-specific efficient algorithms designed. This topic is beyond the scope of this paper and is a subject for future work.

Although not used directly by the JQPG algorithms, the order-based CPG cost functions $Cost_{ord}$ and $Cost^{lat}_{ord}$ (that we will introduce in Section 5.1) also have the ASI property. We formally prove this in the extended paper [27].

4. JQPG FOR GENERAL PATTERN TYPES

The CPG-JQPG reduction presented above only applies to pure conjunctive patterns. However, real-world patterns are much more diverse. To complete the solution, we have to consider simple patterns containing SEQ, OR, NOT and KL operators. We also have to address nested patterns.

This section describes how a pattern of each of the aforementioned types can be represented and detected as either a pure conjunctive pattern or their union. The transformations presented below are only applied for the purpose of plan generation, that is, no actual conversion takes place during evaluation. The formal correctness proofs for the shown reductions can be found in the extended paper [27].

Sequence patterns. We observe that a sequence pattern is merely a conjunctive pattern with additional temporal constraints, i.e., predicates on the values of the timestamp attribute. Thus, a general pure sequence pattern of the form

PATTERN SEQ $(T_1\, e_1, T_2\, e_2, \cdots, T_n\, e_n)$
WHERE $(c_{1,1} \wedge c_{1,2} \wedge \cdots \wedge c_{n,n-1} \wedge c_{n,n})$

can be rewritten as follows without any semantic change:

PATTERN AND $(T_1\, e_1, T_2\, e_2, \cdots, T_n\, e_n)$
WHERE $(c_{1,1} \wedge \cdots \wedge c_{n,n} \wedge$
$\wedge (e_1.ts < e_2.ts) \wedge \cdots \wedge (e_{n-1}.ts < e_n.ts))$.

An instance of the sequence pattern is thus reduced from CPG to JQPG similarly to a conjunctive pattern, with the *timestamp* column added to each relation R_i representing an event type T_i, and constraints on the values of this column introduced into the query representation.

Kleene closure patterns. In a pattern with an event type T_i under a KL operator, any subset of events of T_i within the time window can participate in a match. During plan generation, we are interested in modeling this behavior in a way comprehensible by a JQPG algorithm, that is, using an equivalent pattern without Kleene closure. To that end, we introduce a new type T_i' to represent all event subsets accepted by $KL(T_i)$, that is, the power set of events of T_i. A set of k events of type T_i will be said to contain 2^k "events" of type T_i', one for each subset of the original k events. The new pattern is constructed by replacing $KL(T_i)$ with T_i'. Since a time window of size W contains $2^{r_i \cdot W}$ subsets of T_i (where r_i is the arrival rate of T_i), the arrival rate r_i' of T_i' is set to $\frac{2^{r_i \cdot W}}{W}$. The predicate selectivities remain unchanged.

For example, given the following pattern with the arrival rate of 5 events per second for each event type:

```
PATTERN AND(A a,KL(B b),C c)
WHERE (true) WITHIN 10 seconds,
```

the pattern to be utilized for plan generation will be:

```
PATTERN AND(A a,B' b,C c)
WHERE (true) WITHIN 10 seconds.
```

The arrival rate of B' will be calculated as $r_B' = \frac{2^{r_B \cdot W}}{W} = \frac{1}{10} \cdot 2^{50}$. A plan generation algorithm will then be invoked on the new pattern. Due to an extremely high arrival rate of B', its processing will likely be postponed to the latest step in the plan, which is also the desired strategy for the original pattern in this case. B' will then be replaced with B in the resulting plan, and the missing Kleene closure operator will be added in the respective stage (by modifying an edge type for a NFA [29] or a node type for a tree [36]), thus producing a valid plan for detecting the original pattern.

Negation patterns. Patterns with a negated event will not be rewritten. Instead, we will introduce a negation-aware evaluation plan creation strategy. First, a plan will be generated for a positive part of a pattern as described above. Then, a check for the appearance of a negated event will be added at the earliest point possible, when all positive events it depends on are already received. This construction process will be implemented by augmenting a plan with a transition to the rejecting state for a NFA [29] or with a NSEQ node for a ZStream tree [36]. For example, given a pattern `SEQ(A,NOT(B),C,D)`, the existence of a matching B in the stream will be tested immediately after the latest of A and C have been accepted. Since both Lazy NFA and ZStream incorporate event buffering, this technique is feasible and easily applicable.

Nested patterns. Patterns of this type can contain an unlimited number of n-ary operators. After transforming SEQ to AND as shown above, we are left with only two such operator types, AND and OR. Given a nested pattern, we convert the pattern formula to DNF form, that is, an equivalent nested disjunctive pattern containing a list of simple conjunctive patterns is produced. Then, a separate evaluation plan is created for each conjunctive subpattern, and their detection proceeds independently. The returned result is the union of all subpattern matches.

Note that applying the DNF transformation can cause some expressions to appear in multiple subpatterns. For example, a nested pattern of the form `AND(A,B,OR(C,D))` will be converted to a disjunction of conjunctive patterns `AND(A,B,C))` and `AND(A,B,D))`. As a result, redundant computations will be performed by automata or trees corresponding to different subpatterns (e.g., comparing A's to B's). This problem can be solved by applying known multi-query techniques [17, 35, 43, 44, 54].

5. ADAPTING JQPG ALGORITHMS TO COMPLEX EVENT PROCESSING

The theoretical results from previous sections imply that any existing technique for determining a close-to-optimal execution plan for a join query can be adapted and used in CEP applications. However, many challenges arise when attempting to perform this transformation procedure in practice. First, despite the benefits of the cost function introduced in Section 2.4, simply counting the partial matches is not always sufficient. Additional performance metrics, such as the latency, are often essential. Second, complex event specification languages contain various constructs not present in traditional databases, such as event selection strategies. In this section, we will show how these extensions can be incorporated into existing JQPG algorithms.

In addition, the arrival rates of event types and the predicate selectivities are rarely obtained in advance and can change rapidly over time. An *adaptive* solution must be devised to measure the desired statistics on-the-fly and adapt the evaluation plan accordingly [9, 19, 30, 36]. Due to the considerable importance and complexity of adaptive CEP, we devote a separate paper [28] to discuss this problem.

5.1 Pattern Detection Latency

Latency is defined as the difference between the arrival time of the last event comprising a full match and the time of reporting this match. As many existing applications involve strong real-time requirements, pattern detection latency is a popular optimization goal. Unfortunately, in most cases it is impossible to simultaneously achieve maximal throughput and minimal latency. Trade-offs between the two are widely studied in the CEP context [6, 52].

Detection schemes utilizing out-of-order evaluation, like those discussed in this paper, often suffer from increased latency as compared to simpler approaches. The main reason is that, when an execution plan is optimized for maximal throughput, the last event in the pattern may not be the last event in the plan. After this event is accepted, the system still needs to process the remaining part of the plan, resulting in late detection of the full match.

Algorithms adopted from JQPG do not naturally support latency. However, since they are generally independent of the cost model, this problem can be solved by providing an appropriate cost function. In addition to functions presented in Sections 3.1 and 3.2, which we will refer to as $Cost_{ord}^{trpt}$ and $Cost_{tree}^{trpt}$, a new pair of functions, $Cost_{ord}^{lat}$ and $Cost_{tree}^{lat}$, will reflect the expected latency of a plan. To combine the functions, many existing multi-objective query optimization techniques can be used, e.g., pareto optimal plan calculation [6] or parametric methods [49]. Systems with limited computational resources may utilize simpler and less expensive solutions, such as defining the total cost function as a weighted sum of its two components:

$$Cost\left(Plan\right) = Cost^{trpt}\left(Plan\right) + \alpha \cdot Cost^{lat}\left(Plan\right),$$

where α is a user-defined parameter adjusted to fit the required throughput-latency trade-off. This latter model was used during our experiments (Section 6).

We will now formally define the latency cost functions. For a sequence pattern, let T_n denote the last event type in the order induced by the pattern. Then, for an order-based plan O, let $Succ_O\left(T_n\right)$ denote the event types succeeding T_n in O. Following the arrival of an event of type T_n, in the worst case we need to examine all locally buffered events of types in $Succ_O\left(T_n\right)$. There are $W \cdot r_i$ such events of type T_i, hence $Cost_{ord}^{lat}\left(O\right) = \sum_{T_i \in Succ_O\left(T_n\right)} W \cdot r_i$.

Similarly, for a tree-based plan T, let $Anc_T\left(T_n\right)$ denote all ancestor nodes of the leaf corresponding to T_n in T, i.e., nodes located on a path from T_n to the root (excluding the root). Let us examine the traversal along this path. When an internal node N with two children L and R receives a partial match from, say, the child L, it compares this match to all partial matches currently buffered on R. Thus, the worst-case detection latency of a sequence pattern ending with T_n is proportional to the number of partial matches buffered on the siblings of the nodes in $Anc_T\left(T_n\right)$. More formally, let $sibling\left(N\right)$ denote the other child of the parent of N (for the root this function will be undefined). Then, $Cost_{tree}^{lat}\left(T\right) = \sum_{N \in Anc_T\left(T_n\right)} PM\left(sibling\left(N\right)\right)$.

For a conjunctive pattern, estimating the detection latency is a more difficult problem, as the last arriving event is not known in advance. One possible approach is to introduce a new system component, called the output profiler. The output profiler examines the full matches reported as output and records the most frequent temporal orders in which primitive events appear. Then, as enough information is collected, the latency function may be defined as in the previous case, subject to the event arrival order with the highest probability of appearance.

Finally, for a disjunctive pattern, we define the latency cost function as the maximum over the disjunction operands. This definition applies also for arbitrary nested patterns.

5.2 Event Selection Strategies

In addition to event types, operators and predicates, CEP patterns are further defined using the *event selection strategies* [5, 16, 21]. An event selection strategy specifies how events are selected from an input stream for partial matches. In this section, we discuss four existing strategies and show how a reduction from JQPG to CPG can support them.

Until now, we have implicitly assumed the *skip-till-any-match* selection strategy [5], which permits a primitive event to participate in an unlimited number of matches. This strategy is the most flexible, as it allows all possible combinations of events comprising a match to be detected. However, some streaming applications do not require such functionality. Thus, additional strategies were defined.

The *skip-till-next-match* selection strategy [5] limits an event to appear in at most a single full match. This is enforced by "consuming" events already assigned to a match. While this strategy prevents some matches from being discovered, it considerably simplifies the detection process. In a system operating under skip-till-next-match, our cost model will no longer provide a correct estimate for a number of partial matches. However, since most JQPG algorithms do not depend on a specific cost function, we can solve this issue by replacing $Cost_{ord}$ and $Cost_{tree}$ with newly devised models.

Let us examine the number of partial matches in an order-based setting under the skip-till-next-match strategy. We will denote by $m\left[k\right]$ the number of matches of size k expected to exist simultaneously in a time window. Obviously, $m\left[1\right] = W \cdot r_{p_1}$, where T_{p_1} is the first event type in the selected evaluation order. For the estimate of $m\left[2\right]$, there are two possibilities. If $r_{p_1} > r_{p_2}$, there will not be enough instances of T_{p_2} to match all existing instances of T_{p_1}, and some of the existing matches of size 1 will never be extended. Hence, $m\left[2\right] = W \cdot r_{p_2}$ in this case. Otherwise, as an existing partial match cannot be extended by more than a single event of type T_{p_2}, $m\left[1\right]$ will be equal to $m\left[2\right]$. In addition, if a mutual condition exists between T_{p_1} and T_{p_2}, the resulting expression has to be multiplied by sel_{p_1,p_2}.

By extending this reasoning to an arbitrary partial match, we obtain the following expression:

$$m\left[k\right] = W \cdot min\left(r_{p_1}, r_{p_2}, \cdots, r_{p_k}\right) \cdot \prod_{i,j \leq k; i \leq j} sel_{p_i,p_j}; 1 \leq k \leq n.$$

And the new cost function for order-based CPG is

$$Cost_{ord}^{next}\left(O\right) = \sum_{k=1}^{n}\left(W \cdot m\left[k\right]\right).$$

Using similar observations, the above result can be trivially extended for the tree-based model.

The two remaining selection strategies, *strict contiguity* and *partition contiguity* [5], further restrict the appearance of events in a match. The strict contiguity requirement forces the selected events to be contiguous in the input stream, i.e., it allows no other events to appear in between. The partition contiguity strategy is a slight relaxation of the above. It partitions the input stream according to some condition and only requires the events located in the same partition to be contiguous.

We will base the cost models of these strategies on the one presented above for skip-till-next-match. To express strict contiguity, we will augment each event with an attribute reflecting its position in the stream. Then, we will add a condition for each pair of potentially neighboring events, requiring the numbers to be adjacent. For partition contiguity, the new attribute will represent an inner, per-partition order rather than a global one. The new contiguity condition will first compare the partition IDs of the two events, and only verify their positional counters if the IDs match. We assume that the value distribution across the partitions remains unchanged. Otherwise, the evaluation plan is to be generated on a per-partition basis. Techniques incorporating per-partition plans are beyond the scope of this paper and are a subject for our future research.

6. EXPERIMENTAL EVALUATION

In this section, we present our experimental study on real-world data. Our main goal was to compare some of the well-known JQPG algorithms, adapted for CPG as described above, to the currently used methods developed directly for CPG. The results demonstrate the superiority of the former in terms of quality and scalability of the generated plans.

6.1 CPG and JQPG Algorithms

We implemented 5 order-based and 3 tree-based CPG algorithms. Out of those, 3 order-based and 2 tree-based algorithms are JQPG methods adapted to the CEP domain. The rest are native CPG techniques. The order-based plan generation algorithms included the following:

• Trivial order (TRIVIAL) - the evaluation plan is set to the initial order of the sequence pattern. This strategy is used in various CEP engines based on NFAs, such as SASE [51] and Cayuga [18].

• Event frequency order (EFREQ) - the events are processed by the ascending order of their arrival frequencies. This is the algorithm of choice for frameworks such as PB-CED [6] and the Lazy NFA [30].

• Greedy cost-based algorithm (GREEDY) [48] - this greedy heuristic algorithm for JQPG proceeds by selecting at each step the relation which minimizes the value of the cost function. Here and below, unless otherwise stated, we will use cost functions minimizing the intermediate results size (Sections 3.1 and 3.2).

• Iterative improvement algorithm (II-RANDOM / II-GREEDY) [48] - a local search JQPG algorithm, starting from some initial execution plan and attempting a set of moves to improve the cost function, until a local minimum is reached. We experimented with two variations of this algorithm. The first, denoted as II-RANDOM, starts from a random order. The second, denoted as II-GREEDY, first applies a greedy algorithm to create an initial state. In both cases, the functions used to traverse between states are *swap* (the positions of two event types in a plan are swapped) and *cycle* (the positions of three event types are shifted).

• Dynamic programming algorithm for left-deep trees (DP-LD) [46] - this exponential-time algorithm utilizes dynamic programming to produce a provably optimal execution plan. The result is limited to a left-deep tree topology.

For the tree-based plan generation algorithms, the following were used:

• ZStream plan generation algorithm (ZSTREAM) [36] - creates an evaluation tree by iterating over all possible tree topologies for a given sequence of leaves.

• ZStream with greedy cost-based ordering (ZSTREAM-ORD) - as was demonstrated in Section 2.3, the limitation of the ZStream algorithm is in its inability to modify the order of tree leaves. This algorithm attempts to utilize an order-based JQPG method to overcome this drawback. It operates by first executing GREEDY on the leaves of the tree to produce a 'good' ordering, then applying ZSTREAM on the resulting list.

• Dynamic programming algorithm for bushy trees (DP-B) [46] - same as DP-LD, but without the topology restriction.

6.2 Experimental Setup

The data used during the experiments was taken from the NASDAQ stock market historical records [1]. Each data

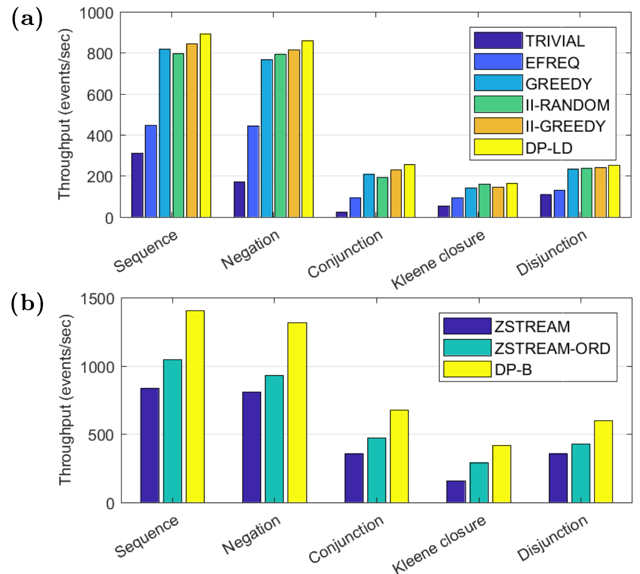

Figure 3: Throughput for different pattern types (higher is better): (a) order-based methods; (b) tree-based methods.

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. For each identifier, a separate event type was defined. We also augmented the event format with the precalculated difference between the current and the previous price of each stock.

The majority of the experiments were performed separately on 5 sets of patterns: (1) pure sequences; (2) sequences with a negated event (marked as 'negation' patterns in the graphs below); (3) conjunctions; (4) sequences containing an event under KL operator (marked as 'Kleene closure' patterns); (5) composite patterns, consisting of a disjunction of three sequences (marked as 'disjunction' patterns). Each set contained 500 patterns with the sizes (numbers of the participating events) ranging from 3 to 7, 100 patterns for each value. The pattern time window was set to 20 minutes.

The pattern structure was motivated by the problem of monitoring the relative changes in stock prices. Each pattern included a number of predicates, roughly equal to half the size of a pattern, comparing the *difference* attributes of two of the involved event types. For example, one pattern of size 3 from the set of conjunctions was defined as follows:

```
PATTERN AND(MSFTStock m, GOOGStock g, INTCStock i)
WHERE (m.difference < g.difference)
WITHIN 20 minutes.
```

All arrival rates and predicate selectivities were calculated during the preprocessing stage. The measured arrival rates varied between 1 and 45 events per second, and the selectivities ranged from 0.002 to 0.88. As discussed in Section 5, in most real-life scenarios these statistics are not available in advance and may fluctuate frequently and significantly during runtime. We experimentally study the impact of these issues in a separate paper [28].

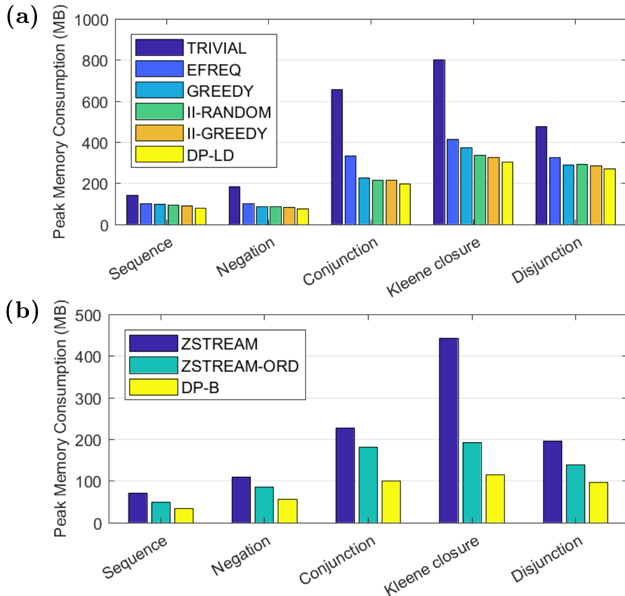

Figure 4: Memory consumption for different pattern types (lower is better): (a) order-based methods; (b) tree-based methods.

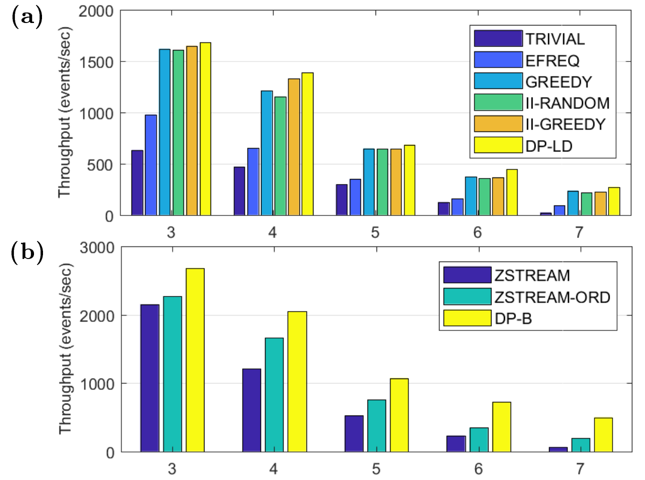

Figure 5: Throughput as a function of the sequence pattern length (higher is better): (a) order-based methods; (b) tree-based methods.

To compare a set of plan generation algorithms, we implemented two evaluation mechanisms discussed in this paper, the out-of-order lazy NFA [30] and the instance-based tree model based on ZStream [36] as presented in Section 2.3. The former was then used to evaluate plans created by each order-based CPG or JQPG algorithm on the patterns generated as described below. The latter was similarly used for comparing tree-based plans.

We selected throughput and memory consumption as our performance metrics for this study. Throughput was defined as the number of primitive events processed per second during pattern detection using the selected plan. To estimate the memory consumption, we measured the peak memory required by the system during evaluation. The metrics were acquired separately for each pattern, and the presented results were then calculated by taking the average.

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 and took about 2.5 months to complete.

6.3 Experimental Results

Figures 3 and 4 present the comparison of the plan generation algorithms described in Section 6.1 in terms of throughput and memory consumption, respectively. Each group represents the results obtained on a particular set of patterns described above, and each bar depicts the average value of a performance metric for a particular algorithm. For clarity, order-based and tree-based methods are shown separately.

On average, the plans generated using JQPG algorithms achieve a considerably higher throughput than those created using native CPG methods. For order-based plans, the perceived gain of the best-performed DP-LD over EFREQ ranged from a factor of 1.7 for iteration patterns to 2.7 for conjunctions. Similar results were obtained for tree-based plans (ZSTREAM vs. DP-B). JQPG methods also display better overall memory utilization. The order-based JQPG plans consume about 65-85% of the memory required by those produced by EFREQ. An even greater difference was observed for tree-based plans, with DP-B using up to almost 4 times less memory than the CEP-native ZSTREAM.

Unsurprisingly, the best performance was observed for plans created using the exhaustive algorithms based on dynamic programming, namely DP-LD and DP-B. However, due to the exponential complexity of these algorithms, their use in practice may be problematic for large patterns, especially in systems where new evaluation plans are to be generated with high frequency. Thus, one goal of the experimental study was to test the exhaustive JQPG methods against the nonexhaustive ones (such as GREEDY and II algorithms) to see whether the performance gain of the former category is worth the high plan generation cost.

For the order-based case, the answer is indeed negative, as the results for DP-LD and the heuristic JQPG algorithms are comparable and no significant advantage is achieved by the former. Due to the relatively small size of the left-deep tree space, the heuristics usually succeed in locating the globally optimal plan. Moreover, the II-GREEDY algorithm generally produces plans that are slightly more memory-efficient. This can be attributed to our cost model, which only counts the partial matches, but does not capture the other factors such as the size of the buffered events. The picture looks entirely different for the tree-based methods, where DP-B displays a convincing advantage over both the basic ZStream algorithm and its combination with the greedy heuristic method.

Another important conclusion from Figures 3 and 4 is that methods following the tree-based model greatly outperform the order-based ones, both in throughput and memory consumption. This is not a surprising outcome, as the tree-based algorithms are capable of creating a significantly larger space of plans. However, the best order-based JQPG algorithm (DP-LD) is comparable or even superior to the CPG-native ZStream in most settings.

Figure 5 depicts the throughput as a function of the pattern size. For lack of space, we only present here the evaluation performed on sequence patterns. The results obtained

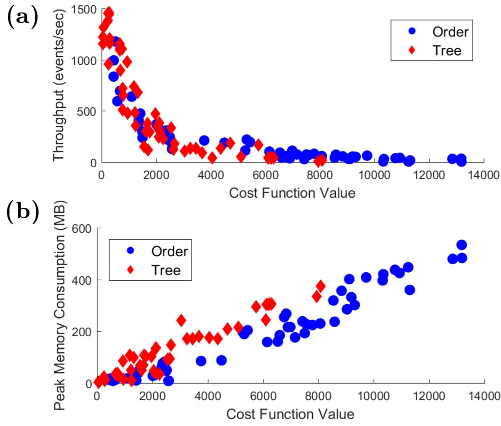

Figure 6: Performance metrics as a function of the cost for order-based and tree-based patterns: (a) throughput; (b) memory consumption.

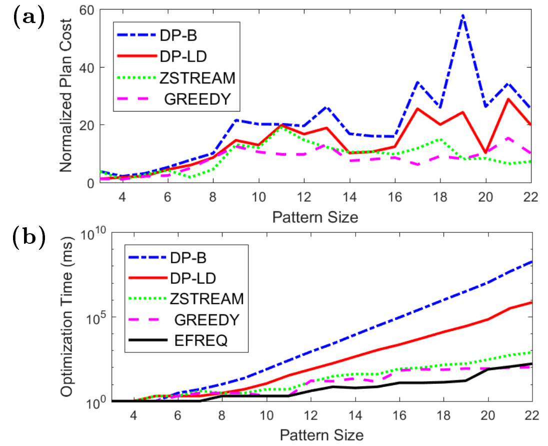

Figure 7: Generation of large plans (selected algorithms): (a) average normalized plan cost (higher is better); (b) average plan generation time (logarithmic scale, lower is better). The results are presented as a function of pattern size.

for the rest of the pattern sets, as well as the respective memory consumption measurements, follow similar trends and are discussed in the extended paper [27]. Although the performance of all methods degrades drastically as the pattern size grows, the relative throughput gain for JQPG methods over native CPG methods is consistently higher for longer sequences. This is especially evident for the tree-based variation of the problem (5(b)), where the most efficient JQPG algorithm (DP-B) achieves 7.6 times higher throughput than the native CPG framework (ZSTREAM) for patterns of length 7, compared to a speedup of only 1.2 times for patterns of 3 events. We can thus conclude that, at least for the pattern sizes considered in this study, the JQPG methods provide a considerably more scalable solution.

In our next experiment, we evaluated the quality of the cost functions used during plan generation. To that end, we created 60 order-based and 60 tree-based plans for patterns of various types using different algorithms. The plans were then executed on the stock dataset. The throughput and the memory consumption measured during each execution are shown in Figure 6 as the function of the cost assigned to each plan by the corresponding function ($Cost_{ord}$ or $Cost_{tree}$). The obtained throughput seems to be inversely proportional to the cost, behaving roughly as $\frac{1}{x^c}; c \geq 1$. For memory consumption, an approximately linear dependency can be observed. These results match our expectations, as a cheaper plan is supposed to yield better performance and require less memory. We may thus conclude that the costs returned by $Cost_{ord}$ and $Cost_{tree}$ provide a reasonably accurate estimation of the actual performance of a plan.

The above conclusion allowed us to repeat the experiments summarized in Figure 5 for larger patterns, using the plan cost as the objective function. We generated 200 patterns of sizes ranging from 3 to 22. We then created a set of plans for each pattern using different algorithms and recorded the resulting plan costs. Due to the exponential growth of the cost with the pattern size, directly comparing the costs was impractical. Instead, the *normalized cost* was calculated for every plan. The normalized cost of a plan Pl created by an algorithm A for a pattern P was defined as the cost of a plan generated for P by the empirically worst algorithm (the CEP-native EFREQ), divided by the cost of Pl.

The results for selected algorithms are depicted in Figure 7(a). Each data point represents an average normalized cost for all plans of the same size created by the same algorithm. As we observed previously, the DP-based join algorithms consistently produced significantly cheaper plans (up to a factor of 57) than the heuristic alternatives. Also, the worst JQPG method (GREEDY) and the best CPG method (ZSTREAM) produced plans of similar quality, with the former slightly overperforming the latter for larger pattern sizes. The worst-performing EFREQ algorithm was used for normalized cost calculation and is thus not shown.

Figure 7(b) presents the plan generation times measured during the above experiment. The results are displayed in logarithmic scale. While all algorithms incur only negligible optimization overhead for small patterns, it grows rapidly for methods based on dynamic programming (for a pattern of length 22, it took over 50 hours to create a plan using DP-B). This severely limits the applicability of the DP-based approaches when the number of events in a pattern is high. On the other hand, all non-DP algorithms were able to complete in under a second even for the largest tested patterns. The join-based greedy algorithm (GREEDY) demonstrated the best overall trade-off between optimization time and quality.

Next, we studied the performance of the hybrid throughput-latency cost model introduced in Section 5.1. Each of the 6 JQPG-based methods discussed in Section 6.1 was evaluated using three different values for the throughput-latency trade-off parameter α: 0, 0.5 and 1. Note that for the first case ($\alpha = 0$) the resulting cost model is identical to the one defined in Section 3 and used in the experiments above. For each algorithm and for each value of α, the throughput and the average latency (in milliseconds) were measured.

Figure 8 demonstrates the results, averaged over 500 patterns included in the sequence pattern set. Measurements obtained using the same algorithm are connected by straight lines, and the labels near the highest points (diamonds) indicate the algorithms corresponding to these points. It can be seen that increasing the value of α results in a significantly lower latency. However, this also results in a considerable drop in throughput for most algorithms. By fine-tuning this parameter, the desired latency can be achieved with min-

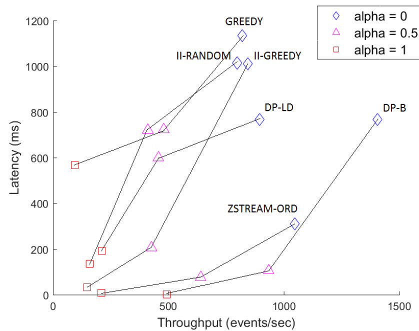

Figure 8: Throughput vs. latency using different values for the alpha parameter of the cost model.

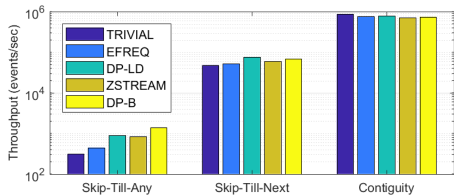

Figure 9: Throughput for different event selection strategies (logarithmic scale).

imal loss in throughput. It can also be observed that the tree-based algorithms DP-B and ZSTREAM-ORD achieve a substantially better throughput-latency trade-off as compared to other methods.

Finally, we performed a comparative throughput evaluation of the sequence pattern set under three different event selection strategies: *skip-till-any-match*, *skip-till-next-match* and *contiguity* (Section 5.2). The results are depicted in Figure 9 for selected algorithms. Due to large performance gaps between the examined methods, the results are displayed in logarithmic scale. For *skip-till-next-match*, JQPG methods hold a clear advantage, albeit less significant than the one demonstrated above for *skip-till-any-match*. The opposite observation can be made about the *contiguity* strategy, where the trivial algorithm following a static plan outperforms other, more complicated methods. Due to the simplicity of the event detection process and the lack of nondeterminism in this case, the plan set by an input specification always performs best, while the alternatives introduce a slight additional overhead of reordering and event buffering.

7. RELATED WORK

Systems for scalable detection of complex events have become an increasingly important research field during last decades [16, 21]. Their inception can be traced to earlier systems for massive data stream processing [3, 8, 11, 12]. Later, a broad variety of general purpose complex event processing solutions emerged [4, 6, 10, 15, 18, 30, 34, 36, 42, 45, 51], including the widely used commercial CEP providers, such as Esper [2] and IBM System S [7].

Various performance optimization techniques are implemented in CEP systems [23]. In [42], a rewriting framework is described, based on unifying and splitting patterns. A method for efficient Kleene closure evaluation based on sharing with postponed operators is discussed in [53], while in [41] the above problem is solved by maintaining a compact graph encoding of event sequences and utilizing it for effective reuse. RunSAT [20] utilizes another approach, preprocessing a pattern and setting optimal points for termination of the detection process. ZStream [36] presents an optimization framework for optimal tree generation, based on a complex cost model. Advanced methods were also proposed for multi-query CEP optimization [17, 35, 43, 44, 54].

CEP engines utilizing the order-based evaluation approach have also adopted different optimization strategies. SASE [51], Cayuga [18] and T-Rex [15] design efficient data structures to enable smart runtime memory management. These NFA-based mechanisms do not support out-of-order processing, and hence are still vulnerable to the problem of large intermediate results. In [6, 30, 45], various pattern reordering methods for efficient order-based complex event detection are described. None of these works takes the selectivities of the event constraints into account.

Calculating an optimal evaluation plan for a join query has long been considered one of the most important problems in the area of query optimization [47]. Multiple authors have shown the NP-completeness of this problem for arbitrary query graphs [13, 24], and a wide range of methods were proposed to provide either exact or approximate close-to-optimal solutions [31, 32, 33, 38, 46, 47, 48].

Methods for join query plan generation can be roughly divided into two main categories. The heuristic algorithms produce fast solutions, but the resulting execution plans are often far from the optimum. They are often based on combinatorial [25, 47, 48] or graph-based [32, 33] techniques.

The second category, the exhaustive algorithms, provide provable guarantees on the optimality of the returned solutions. These methods are often based on dynamic programming [38, 46] and thus suffer from worst-case exponential complexity. Hybrid techniques presenting a trade-off between the speed of heuristic approaches and the precision of DP-based approaches were also proposed [31].

Incorporating join optimization techniques from traditional DBMSs was already considered in the fields related to CEP, such as XPath [22] and data stream processing [12, 23]. To the best of our knowledge, none of the above works provides a formal reduction to JQPG.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the relationship between CEP Plan Generation and Join Query Plan Generation. It was shown that the CPG problem is equivalent to JQPG for a subset of pattern types, and reducible to it for other types. We discussed how close-to-optimal solutions to CPG can be efficiently obtained in practice by applying existing JQPG methods. The presented experimental evaluation results supported our theoretical analysis. Our future work will target advanced challenges of applying join-related techniques in the field of CEP, such as handling predicate dependencies and data uncertainty.

9. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 Research And Innovation Programme under grant agreement no. 688380 and was partially supported by the Israel Science Foundation (grant no. 919191) and by HPI-Technion Research School.

10. REFERENCES

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

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

[3] 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.

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

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2006.

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

[7] 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.

[8] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[9] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 407–418, New York, NY, USA, 2004. ACM.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS*, pages 50–61. ACM, 2010.

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

[16] 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.

[17] 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.

[18] 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.

[19] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, January 2007.

[20] 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.

[21] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[22] T. Grust, J. Rittinger, and J. Teubner. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 949–958, New York, NY, USA, 2007. ACM.

[23] 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.

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

[25] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321, May 1990.

[26] Y. Ioannidis and Y. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *SIGMOD Rec.*, 20(2):168–177, April 1991.

[27] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. *CoRR*, abs/1801.09413, 2017.

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

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

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

[31] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[32] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 128–137, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[33] C. Lee, C.S. Shih, and Y.H. Chen. A graph-theoretic model for optimizing large join queries. In *DASFAA*, volume 6 of *Advanced Database Research and Development Series*, pages 87–96. World Scientific, 1997.

[34] 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.

[35] 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.

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

[37] G. Moerkotte. Building query compilers. 2014. http://pi3.informatik.uni-mannheim.de/ moer/querycompiler.pdf.

[38] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 930–941. VLDB Endowment, 2006.

[39] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Math. Oper. Res.*, 4(3):215–224, August 1979.

[40] M. Orlowski. On optimization of joins in distributed database system. In *Future Databases 92*, pages 106–114. 1992.

[41] 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.

[42] 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.

[43] 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. ACM.

[44] 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*, EDBT '13, pages 525–536, New York, NY, USA, 2013. ACM.

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

[46] 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.

[47] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

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

[49] I. Trummer and C. Koch. Multi-objective parametric query optimization. *SIGMOD Rec.*, 45(1):24–31, 2016.

[50] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. *SIGMOD Rec.*, 25(2):35–46, June 1996.

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

[52] I. Yi, J. G. Lee, and K. Y. Whang. Apam: Adaptive eager-lazy hybrid evaluation of event patterns for low latency. In *Proceedings of the 25th ACM Conference on Information and Knowledge Management*, pages 2275–2280. ACM, 2016.

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

[54] 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.