High-Performance Nested CEP Query Processing over Event Streams

Mo Liu [#], Elke Rundensteiner [#], Dan Dougherty [#], Chetan Gupta [*], Song Wang [*], Ismail Ari , Abhay Mehta [*]

[#] *Worcester Polytechnic Institute, USA*
{liumo|rundenst|dd}@cs.wpi.edu

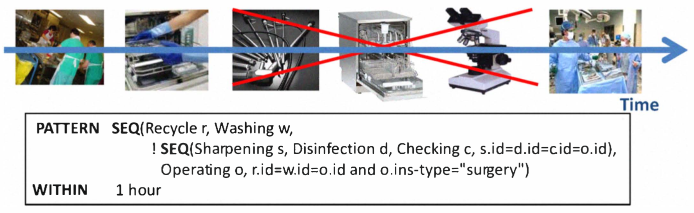
[*] *Hewlett-Packard Labs, USA*
{chetan.gupta|songw|abhay.mehta}@hp.com

Ozyegin University, Turkey
Ismail.Ari@ozyegin.edu.tr

***Abstract*— Complex event processing (CEP) over event streams has become increasingly important for real-time applications ranging from health care, supply chain management to business intelligence. These monitoring applications submit complex queries to track sequences of events that match a given pattern. As these systems mature the need for increasingly complex nested sequence query support arises, while the state-of-art CEP systems mostly support the execution of flat sequence queries only. To assure real-time responsiveness and scalability for pattern detection even on huge volume high-speed streams, efficient processing techniques must be designed. In this paper, we first analyze the prevailing nested pattern query processing strategy and identify several serious shortcomings. Not only are substantial subsequences first constructed just to be subsequently discarded, but also opportunities for shared execution of nested subexpressions are overlooked. As foundation, we introduce *NEEL*, a CEP query language for expressing nested CEP pattern queries composed of sequence, negation, AND and OR operators. To overcome deficiencies, we design rewriting rules for pushing negation into inner subexpressions. Next, we devise a normalization procedure that employs these rules for flattening a nested complex event expression. To conserve CPU and memory consumption, we propose several strategies for efficient shared processing of groups of normalized *NEEL* subexpressions. These strategies include prefix caching, suffix clustering and customized "bit-marking" execution strategies. We design an optimizer to partition the set of all CEP subexpressions in a *NEEL* normal form into groups, each of which can then be mapped to one of our shared execution operators. Lastly, we evaluate our technologies by conducting a performance study to assess the CPU processing time using real-world stock trades data. Our results confirm that our *NEEL* execution in many cases performs 100 fold faster than the traditional iterative nested execution strategy for real stock market query workloads.**

I. Introduction

Complex event processing (CEP) has become increasingly important in modern applications, ranging from supply chain management for RFID tracking to real-time intrusion detection [1], [2], [3]. CEP must be able to support sophisticated pattern matching on real time event streams including the arbitrary nesting of sequence (SEQ), AND, OR and the flexible use of negation in such nested patterns. For example, consider reporting contaminated medical equipments in a hospital [4], [5], [6]. Let us assume that the tools for medical operations are RFID-tagged. The system monitors the histories of the equipment (such as, records of surgical usage, of washing, sharpening and disinfection). When a healthcare worker puts a box of surgical tools into a surgical table equipped with RFID readers, the computer would display warnings such as "The tool with id = "5" must be disposed". Query Q_1 (Figure 1) expresses this critical condition that after being recycled and washed, a surgery tool is being put back into use without first being sharpened, disinfected and then checked for quality assurance. Such complex sequence queries may contain complex negation specifying the non-occurrence of composite subpatterns, such as negating the composite event of sharpened, disinfected and checked subsequences.



```
PATTERN  SEQ(Recycle r, Washing w,
             ! SEQ(Sharpening s, Disinfection d, Checking c, s.id=d.id=cid=o.id),
              Operating o, r.id=w.id=o.id and o.ins-type="surgery")
WITHIN   1 hour
```

Fig. 1. Example Query Q_1

One of the most interesting and flexible features of a query language is the nesting of operators to an arbitrary depth [7], [8]. Without this capability, users are severely restricted in forming complex patterns in a convenient and succinct manner. However, the state-of-art CEP systems including SASE [1] and ZStream [3] do not support such nested queries. Even though the Cayuga system [2] mentions composable queries, it only allows sub-queries in the FROM clause and it also doesn't support applying negation over composite event types. Our objective however is to allow the specification of negation within any level of the nested query as shown above in the example. While CEDR [9] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. In short, no processing nor optimization mechanisms for nested CEP queries have been proposed in the literature to date.

Without the design of an optimized execution strategy for nested sequence queries, an iterative nested execution strategy would typically be adopted by default [10], [11], [12]. Namely, first all component events matching the outer query are identified. In our example, we thus would compute all matching composite events consisting of SEQ(Recycle, Washing, Operating) subsequences. Thereafter, for each outer SEQ(Recycle, Washing, Operating) match, the results for the nested inner subsequences are iteratively computed, i.e., in this case, (Sharpening, Disinfection, Checking) subsequences. As last step, each outer candidate sequence result will be filtered by the non-existence of the inner subsequence match between the Washing reading and Operating reading. This process of first rigidly undertaking the construction of sequence results for the outer operators and then constructing sequence results for the inner operators is not efficient as it misses critical opportunities for optimization as we will illustrate below.

Problem 1: Candidate sequence results generated may later simply be discarded – thus wasting precious resources. For example in the above query Q_1, the generation of the sequence results for the outer subexpression SEQ(Recycle, Washing, Operating) may all be wasted as during normal medical procedures inner sequences of type (Sharpening, Disinfection, Checking) would indeed exist between event pairs of Washing and Operating. This unnecessary event generation of later again discarded candidate sequence results wastes precious memory and CPU processing resources.

Problem 2: Full results satisfying the nested negated subexpression, such as instances that match the subsequence SEQ(Sharpening s, Disinfection d, Checking c) in Q_1 will be repeatedly constructed and processed for each outer candidate. However, knowing the existence of only one (Sharpening s, Disinfection d, Checking c) event between Washing and Operating events would be sufficient for filtering a candidate.

Our goal is to design nested CEP processing and optimization strategies that overcome the above identified shortcomings – thus significantly saving CPU processing resources. In this paper, we make the following contributions:

- First we introduce the nested CEP language *NEEL* that supports the flexible nesting of AND, OR, Negation and SEQ operators at any level. We also describe the query algebra of *NEEL*.
- Based on this foundation, we develop a set of equivalence rules for rewriting *NEEL* expressions. Then, we propose a normalization procedure that employs these rewriting rules to transform a nested CEP query into an equivalent non-nested query – thus opening the opportunity for query optimization.
- The normalized expression exposes opportunities for query optimization by shared expression processing. We propose several strategies for implementing physical operators for the shared execution of a set of similar normalized subexpressions, including prefix caching, suffix clustering and a customized "bit-marking" method.
- Due to the exponential search space, we propose an effective cost-based search heuristic for establishing groupings of subexpressions – each then mappable to one of our above shared execution physical operators.
- We thoroughly evaluate our optimized *NEEL* execution through experiments comparing it to the state-of-the-art technique, namely iterative nested execution [11]. Our results confirm that our *NEEL* execution in many cases performs 100 fold faster than the traditional iterative nested execution for real stock market query workloads.

II. Nested CEP Query Model

A. Event Model

An event instance is an occurrence of interest in a system which can be either primitive or composite as further introduced below. *A primitive event instance* denoted by a lowercase letter (e.g.,'e') is the smallest, atomic occurrence of interest in a system. e_i.ts and e_i.te denote the start and the end timestamp of an event instance e, respectively, with e_i.ts $\leq$ e_i.te. For a primitive event instance e_i, e_i.ts = e_i.te. For simplicity, we use the subscript i attached to a primitive instance e to denote the timestamp i. *A composite event instance* is composed of constituent primitive event instances $e = < e_1, e_2, ..., e_n >$. A composite event instance e occurs over an interval. The start and end timestamps of e are equal to e.ts = $\min\{e_i.\text{ts} \mid \forall\ e_i \in \text{e}\}$ and e.te = $\max\{e_i.\text{te} \mid \forall\ e_i \in \text{e}\}$, respectively.

An *event type* is denoted by a capital letter, say E_i. An event type E_i describes a set of attributes that the event instances of this type share. An event type can be either a primitive or a composite event type [13]. *Primitive event types* are predefined in the application domain of interest. *Composite event types* are aggregated event types created by combining other primitive and/or composite event types to form an application specific type. $e_i \in E_j$ denotes that e_i is an instance of the event type E_j. We use e_i.type to denote the type E_j of e_i. Suppose one of the attributes of type E_j is attrj and $e_i \in E_j$, we use e_i.attrj to denote e_i's value for that attribute attrj.

B. NEEL: The Nested Complex Event Language

We now briefly introduce the *NEEL*[1] query language [14] for specifying nested complex event pattern queries. *NEEL* is an extension of non-nested CEP languages from the literature [9], [1], [2]. *NEEL* supports the nesting of AND, OR, Negation and SEQ operators at any level. Q_1 in Figure 1 is a sample query expressed by *NEEL*. For a more detailed discussion as well as several case studies of the *NEEL* language, the reader is referred to [14].

The PATTERN clause retrieves event instances specified in the event expression from the input stream. The qualification in the PATTERN clause further filters event instances by evaluating predicates applied to potential matching events. The WITHIN clause specifies a time period within which all the events of interest must occur in order to be considered a match. In our language, the time period is expressed as a sliding window, though other window semantics could also be applied.

[1]NEEL stands for **Ne**sted Complex **E**vent Query **L**anguage.

```
<Query>::= PATTERN <event-expression>
           WITHIN <window>
           [RETURN <set of primitive events>]
<event-expression> = <ex>
-----------------------------------------------------
<ex> ::=
 SEQ((<ex> | ! (<ex>, [<q>]))*,<ex>, (<ex> |
! (<ex>, [<q>]))*, [<q>])
| AND((<ex>, (<ex> | ! (<ex>, [<q>]))*, [<q>])
| OR((<ex>)+, [<q>])
| (<primitive-event type>, [<var>])
-----------------------------------------------------
<primitive-event type> ::= E1 | E2 | ...
<var> ::= event variable ei
<q>::= (<elemqual>)*
<elemqual> ::= <var>.attr <op> <var>.attr |
    <var>.attr <op> constant
<op> ::= < | > | ≤ | ≥ | = | ! =
<window>::= time duration w | tuple count c
```

TABLE I
NEEL QUERY LANGUAGE

"A set of histories" is returned as a result with each history equal to one "set of instance matches".

Operators in the PATTERN clause. SEQ in the PATTERN clause specifies the particular temporal order in which the event instances of interest should occur. The components of the sequence are the stipulated occurrences and non-occurrences of events of certain event types [15].

Definition 1: [SEQ operator]. SEQ(E_1 e_1 ,..., E_i e_i ,..., E_n e_n) specifies a temporal order in which the event instances of interest e_1 ,..., e_i, ,..., e_n must occur. The output is a composite event e composed of e_1 to e_n such that e_1.ts < ... < e_i.ts < ... < e_n.ts, and e_n.ts - e_1.ts $\leq$ window with the window specified in the WITHIN clause.

Example 1: Given SEQ(*Recycle* r, *Washing* w) and the partial input stream r_1, w_2, w_3 all falling within the window. Then SEQ(*Recycle* r, *Washing* w) generates 2 results $\{r_1, w_2\}$ and $\{r_1, w_3\}$.

Definition 2: [OR operator]. OR operator specifies disjunction of occurrences of events. OR(E_1 e_1 ,..., E_i e_i ,..., E_n e_n) means one or more event instances of types E_1 ,..., E_i ,..., E_n occur within a specified time window.

Definition 3: [AND operator]. AND(E_1 e_1 ,..., E_i e_i ,..., E_n e_n) means event instances of types E_1 ,..., E_i ,..., E_n occur within a specified time window, and their order does not matter. AND operator computes the cross product of input events of the specified types.

Example 2: Given AND(*Recycle* r, *Washing* w) and the partial input stream w_1, r_2, w_3 within the window. Then the two results $\{r_2, w_1\}$ and $\{r_2, w_3\}$ are generated.

Definition 4: [Negation]. The symbol "!" before an event expression E_i expresses the negation of E_i and indicates that E_i is not allowed to appear in the specified position [1].

Any component of SEQ including at the start or the end can be negated using "!". SEQ(E_1 e_1, ! E_2 e_2, E_3 e_3) indicates that e_3 follows e_1 within a specified window without any interleaving instances of e_2 between e_1 and e_3. AND(E_1 e_1, ! E_2 e_2, E_3 e_3) indicates that both e_1 and e_3 occur with no e_2 within the specified window. If there is a ! (Negation) symbol before an event expression, we now say that the event expression marked by ! is a negative event expression. Otherwise it is a positive event expression. At least one positive event expression must exist in SEQ and AND operators.

Example 3: Given AND(*Recycle* r, *Washing* w, ! *Checking* c) and the partial input stream c_1, w_2 and r_3, no results are generated due to the existence of the *Checking* event c within the window.

Nested expression and variable scope. If E_1, E_2 ,..., E_n are event expressions, an application of SEQ, AND and OR over these event expressions is again an event expression [13]. An event expression exp_i can be used as an **inner** component to construct an **outer** expression exp_j. The event instances in an outer expression are visible within the outer expression as well as within the scope of its own nested inner expressions. Q_1 in Figure 1 is an example of a nested expression. The outer expression is SEQ(Recycle r, Washing w, Operating o) and the inner expression is SEQ(Sharpening, Disinfection, Checking). The variables r, w and o in the outer expression are visible in the inner expression.

Predicate specification. The optional qualification [<qual>] in the PATTERN clause contains one or more predicates. Predicates only referring to events in expression exp_i are specified directly inside exp_i (**simple predicates**). Predicates referring to both event instances from the outer and the inner expressions are **correlated predicates**. They must be placed with the innermost expression where a variable used in the expression is declared.

C. NEEL System Overview

Figure 2 shows the *NEEL* system architecture including its core components: Plan-Generator, *NEEL* Rewriter, Plan-Finder and *NEEL* Executor. After a nested CEP query is submitted, the query expressed by a *NEEL* specification is translated into a default nested query plan by the Plan-Generator. We then apply the rewriting procedure (see Section III) to flatten the nested CEP query. Given the set of normalized expressions, the Plan-Finder employs a search method (Section V) to find an optimized shared execution plan considering multiple ways of computation sharing (as will be presented in Section IV). Lastly, the executor instantiates the physical algebra operators according to the plan constructed by the Plan-Finder and then starts continuous CEP execution.

D. Nested CEP Query Plan Generation

A query expressed by a *NEEL* specification is translated into a default nested query plan composed of the following algebraic operators: Window Sequence (*WinSeq*), Window Or (*WinOr*) and Window And (*WinAnd*). The same window w is pushed down and applied to all operator nodes. During query transformation, each expression in the event pattern is mapped to one operator node in the query plan. *WinSeq* first extracts all matches to the positive components specified in the query, and then filters out events based on negative components as specified in the query. *WinOr* returns an event e if e matches

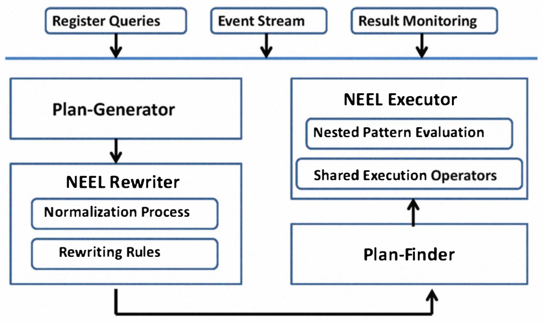


Fig. 2. System Overview

one of the event expressions specified in the WinOr operator. *WinAnd* computes the cross product of its positive components. For queries expressed by *NEEL*, predicates are placed into the proper positions in the respective nested event expressions (see Section II-B).

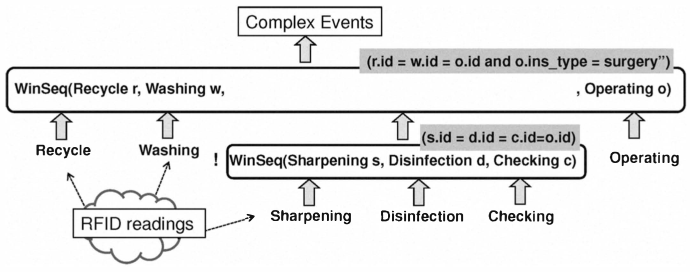

Complex Events
(r.id = w.id = o.id and o.ins_type = surgery")
WinSeq(Recycle r, Washing w,
, Operating o)
Recycle
Washing
!
(s.id = d.id = c.id=o.id)
WinSeq(Sharpening s, Disinfection d, Checking c)
Operating
RFID readings
Sharpening
Disinfection
Checking

Fig. 3. Basic Query Plan

Example 4: Figure 3 shows the query plan for Q_1 in Figure 1. The two SEQ expressions in Q_1 are transformed to two WinSeq operator nodes in the plan. The predicate s.id = d.id = c.id = o.id is placed with the inner WinSeq operator node containing the negative component. The other predicates are attached to the topmost WinSeq operator node.

E. Nested CEP Query Execution

State-of-the-art Stack Based Query Evaluation. We briefly review the implementation strategy of one of the operators, namely, the SEQ operator, while the others are implemented in a similar fashion [11]. We adopt the state-of-art stack-based strategy for execution [1], [16], [17]. An indexing data structure named *SeqState* associates a stack with each event type in the query. Each received event instance is simply appended to the end of the corresponding stack. Event instances are augmented with pointers ptr_i to adjacent events to facilitate the quick locating of related events in other stacks during result construction.

The arrival of an event instance e_m of the last event type E_m of a query q_i triggers the compute function of q_i[2]. The result construction is done by a depth first search along instance pointers ptr_i rooted at that last arrived instance e_m of the event type E_m. All paths composed of edges "reachable" by that root e_m correspond to one matching event sequence returned for q_i. When negative event types are specified in WinSeq, then during sequence construction any edges "reachable" from the root e_m are skipped if an instance of the negative event type is found in the corresponding stream position. Events that are outdated based on the window constraints are purged.

[2]if E_m is a negative event type, postponed sequence evaluation is applied. We omit the details here.

Iterative Nested Execution Strategy. Following the principle of top down iterative query execution for nested SQL queries [18], the outer query is evaluated first and then used as context when evaluating its inner sub-queries. For every outer partial query result, a constrained window is passed down for processing each of its children sub-queries. These sub-queries compute results involving events within the constraint window. Qualified result sequences of the inner operators are passed up to the parent operator and the outer operator then joins its own local results with that of its positive sub-queries. The outer sequence result is filtered if the result set of any of its negative sub-queries is not empty. We apply this iterative execution iteratively until a final result sequence is produced by the root operator or until the process terminates. Finally, the process repeats when the outer query consumes the next instance e. We omit the detailed discussion here for nested queries with negation and predicates due to space constraints. Please refer to [11].

Discussion of Limitation. Such nested query evaluation methodology suffers from several inefficiencies. First, candidate results of SEQ(Recycle r, Washing w, Operating o) initially generated may later need to be discarded. Another potential pitfall is that full results for the negative component SEQ(Sharpening s, Disinfection d, Checking c) are constructed. These cases have also been highlighted as problems 1 and 2 in the introduction. The iterative execution method does not solve these problems [11]. To overcome such inefficiencies, we will explore query rewriting techniques to flatten and optimize nested CEP expressions (Section III).

III. NEEL Event Expression Rewriting

Next, we will present our rewriting procedure for a nested CEP query expressed by *NEEL*.

A. Event Expression Rewriting Rules

Our proposed rewriting rules fall into three categories: flattening rules, distributive rules and negation push down rules. For space reasons, we only briefly discuss some example rules, namely, one for each of the three categories. A complete discussion of the rewriting rules and their correctness based on our *NEEL* semantics can be found in our technical report [19].

Flattening Rule. Table II lists the sample flattening rule for nested CEP expressions. The inner SEQ subexpression is merged into the outer SEQ expression.

Example 5: Given the *NEEL* expression Q_3 = SEQ(E_1, SEQ(E_5, ! E_6, E_7)), after applying the flattening rule in Table II, we get Q_3 = SEQ(E_1, E_5, ! E_6, E_7).

Distributive Rule. Table III lists the sample distributive rule for nested CEP expressions.

SEQ(SEQ(E_1 e_1 ,..., ! (E_i e_i), E_j e_j), ..., E_n e_n)
= SEQ(E_1 e_1 ,..., ! (E_i e_i), E_j e_j ,..., E_n e_n)

TABLE II

SAMPLE FLATTENING RULE.

SEQ(E_1 e_1, OR(E_2 e_2 ,..., E_i e_i), E_j e_j ,..., E_n e_n)
= SEQ(E_1 e_1, E_2 e_2, E_j e_j ,..., E_n e_n) OR ... OR
SEQ(E_1 e_1, E_i e_i, E_j e_j ,..., E_n e_n)

TABLE III

SAMPLE DISTRIBUTIVE RULE.

Example 6: Given the *NEEL* expression Q_4 = SEQ(E_1, E_2 OR SEQ(E_5, ! E_6, E_7)), after applying the distributive rule in Table III, we get Q_4 = SEQ(E_1, E_2) OR SEQ(E_1, SEQ(E_5, ! E_6, E_7)).

Negation Push Down Rule. Table 6 lists one sample negation push down rule for nested CEP expressions. Negation (!) is pushed into the inner SEQ subexpression. The default pattern matching returns all results of a pattern. When Proj exists before SEQ, full results involving these event types listed in Proj are computed and we only check the existence of the positive events not listed in Proj but do not return them.

SEQ(E_1 e_1, ! SEQ(E_2 e_2 ,..., E_{i-1} e_{i-1}, E_i e_i), E_n e_n)
= $Proj_{E_1,E_n}$(SEQ(E_1 e_1, ! (E_i, e_i) OR
SEQ(! (E_{i-1} e_{i-1}), E_i e_i) OR ... OR
SEQ(! (E_2 e_2) ,..., E_{i-1} e_{i-1}, E_i e_i), E_n e_n))

TABLE IV

SAMPLE NEGATION PUSH DOWN RULE.

Example 7: Given the *NEEL* expression Q_5 = SEQ(E_1, ! SEQ(E_2, E_3, E_4), E_5), after applying the negation push down rule in Table 6, we get Q_5 = SEQ(E_1, ! E_4 OR SEQ(! E_3, E_4) OR SEQ(! E_2, E_3, E_4), E_5).

B. Logical Plan: Normal Forms for CEP Expressions

We distinguish between three normal forms for *NEEL* expressions: disjunctive normal form (DNF), conjunctive normal form (CNF) and a nested AND/SEQ expression.

Definition 5: A *NEEL* event expression E is said to be in the **disjunctive normal form** if it is a disjunction of conjuncts say (E_1 OR E_2 OR ... OR E_n) with each query conjunct E_i a sequential pattern specified with one SEQ formed by only primitive event types.
The BNF is:

<event-expression> :: = <E> (OR <E>)*
<E> :: = [Proj((E_i)$^+$)] SEQ((E_i | ! E_i)*, E_i, (E_i | ! E_i)*)

Proj((E_i)$^+$) is syntactic sugar representing the positive event types in the original event expression before rewriting. We omit the formal definition here [19].

Definition 6: A *NEEL* event expression E is said to be in the **conjunctive normal form** if it is a conjunction of disjuncts say (E_1 AND E_2 AND ... AND E_n) with each query disjunct E_i a sequential pattern specified with one SEQ formed by only primitive event types.
The BNF is:

<event-expression> :: = <E> (AND <E>)*
<E> :: = [Proj((E_i)$^+$)] SEQ((E_i | ! E_i)*, E_i, (E_i | ! E_i)*)

When an event expression has nested SEQ and AND operators, we can't completely flatten it into either of the two normal forms in Definitions 5 and 6. As the AND operator doesn't require the ordering among event occurrences while the SEQ operator does, converting the AND operator into the SEQ operator would incur an exponential number of subexpressions. Thus we instead introduce Definition 7 below.

Definition 7: A *NEEL* event expression E is said to be a **nested AND/SEQ expression** if the following properties hold:

Property 1: If an OR operator arises in the expression, then it is the root operator of E.

Property 2: "!" is exclusively before primitive event types.

Property 3: No SEQ (AND) operator is directly nested within another SEQ (AND) operator. However, SEQ can be directly nested within an AND operator, and vice versa.

C. NEEL Expression Flattening Procedure

Our proposed CEP expression flattening procedure illustrated below transforms an arbitrarily nested CEP expression into a *NEEL* normal form as defined in Section III-B above.
Input: An event expression E_{in}.
Output: A normalized expression E_{out} of expression type as in Definitions 5, 6 or 7 (Section III-B).

- *Step 1:* Push ! into expressions recursively until ! is exclusively in front of primitive event expressions by applying the Negation Push Down Rules (Table 6).
- *Step 2:* Apply the Distributive Rules until they are no longer applicable (Table III).
- *Step 3:* Apply Flattening Rules (FR) until no longer applicable (Table II).

Example 8: Given the *NEEL* expression Q_6 = SEQ(E_1, ! SEQ(E_2, E_3, E_4), SEQ(E_5, E_6, E_7))

- By step 1 applying the negation push down rules, we get Q_6 = $Proj_{(E_1,E_5,E_6,E_7)}$(SEQ(E_1, ! E_4 OR SEQ(! E_3, E_4) OR SEQ(! E_2, E_3, E_4), SEQ(E_5, E_6, E_7));
- By step 2 applying distributive rules, we get Q_6 = $Proj_{(E_1,E_5,E_6,E_7)}$(SEQ(E_1, ! E_4, SEQ(E_5, E_6, E_7)) OR SEQ(E_1, SEQ(! E_3, E_4), SEQ(E_5, E_6, E_7)) OR SEQ(E_1, SEQ(! E_2, E_3, E_4), SEQ(E_5, E_6, E_7)));
- By step 3 applying flattening rules, we get Q_6 = $Proj_{(E_1,E_5,E_6,E_7)}$(SEQ(E_1, ! E_4, E_5, E_6, E_7) OR SEQ(E_1, ! E_3, E_4, E_5, E_6, E_7) OR SEQ(E_1, ! E_2, E_3, E_4, E_5, E_6, E_7)). Q_6 is in the disjunctive normal form as defined in Definition 5.

A number of interesting properties can be established about our proposed rewriting procedure. Our rewriting system applied to an event expression is guaranteed to find a normalized form as defined by Definitions 5, 6 and 7. We can show that to apply rewriting rules in different orders in our rewriting procedure doesn't affect the final rewriting result. No infinite rewriting loops will arise, that is our rewriting procedure will stop after finite rewriting steps. We omit precise formulations of these properties as well as these proofs here [19].

IV. Shared Optimized *NEEL* Pattern Execution

Once a normalized expression has been constructed by our rewriting procedure described in Section III-C, multiple sharing opportunities among subexpressions have been exposed. Below, we introduce the strategies we have designed for subexpression sharing among query conjuncts, disjuncts and leaf components[3] in the normalized form as by Definitions 5, 6 and 7.

A. Subexpression Sharing

Sharing with Prefix Caching. First, expressions with a common prefix can share the same cached results. It is wasteful for sequence construction to traverse the same set of stacks repeatedly. Thus the prefix caching method is designed to cache such results in the *PreCache*. This enables future sequence construction involving the same set of stacks to reuse these cached results. The common prefix is computed first before computing each expression. The buffered result e can be deleted savely after an event e_i with e_i.ts - e.ts > window w is received.

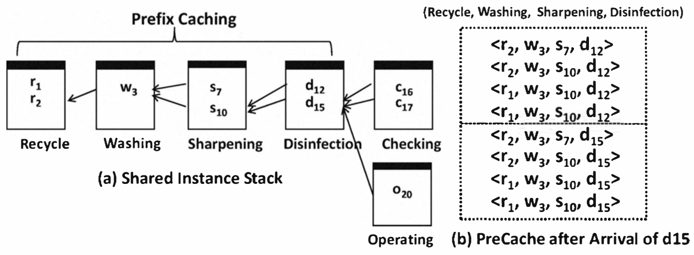

Prefix Caching
r_1 r_2
w_3
s_7 s_{10}
d_{12} d_{15}
c_{16} c_{17}
o_{20}
Recycle
Washing
Sharpening
Disinfection
Checking
Operating
(a) Shared Instance Stack
(Recycle, Washing, Sharpening, Disinfection)
<r_2, w_3, s_7, d_{12}>
<r_2, w_3, s_{10}, d_{12}>
<r_1, w_3, s_{10}, d_{12}>
<r_1, w_3, s_{10}, d_{12}>
<r_2, w_3, s_7, d_{15}>
<r_2, w_3, s_{10}, d_{15}>
<r_1, w_3, s_{10}, d_{15}>
<r_1, w_3, s_{10}, d_{15}>
(b) PreCache after Arrival of d15

Fig. 4. Prefix Caching Example

Example 9: Assume we get a disjunctive normal form with two conjuncts E_1 = SEQ(Recycle, Washing, Sharpening, Disinfection, Checking) and E_2 = SEQ(Recycle, Washing, Sharpening, Disinfection, Operating). Their common prefix is SEQ(Recycle, Washing, Sharpening, Disinfection). To avoid re-constructing results for the common prefix, such shared results (ordered by end timestamps) are stored in *PreCache* as shown in Figure 4. E_1 and E_2 results can then be computed simply by joining the results in the PreCache with events in Checking and Operating stacks respectively.

Sharing with Suffix Clustering. Since event traversals for result construction typically start from events of the last event type in a pattern [1], [20], shared suffices also eliminate redundant event traversals. Queries sharing the same suffices would then be evaluated concurrently by processing their shared suffices until the common part has been treated. Thereafter, each query is finished up by joining the suffix results with other events in the respective query to form final results.

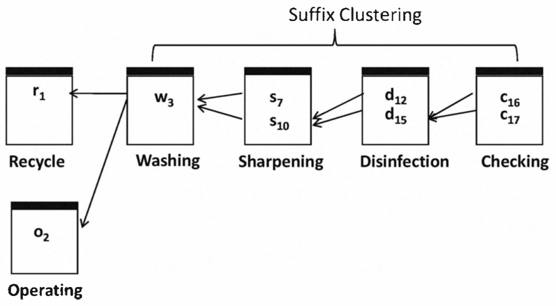


Fig. 5. Suffix Clustering Example

Example 10: Assume we get a conjunctive normal form with two disjuncts E_1 = SEQ(Recycle, Washing, Sharpening, Disinfection, Checking), E_2 = SEQ(Operating, Washing, Sharpening, Disinfection, Checking). Figure 5 shows the stacks shared among E_1 and E_2. Once the event c_{16} or c_{17} of type Checking arrives, the shared result construction for the suffix sub-pattern (Washing, Sharpening, Disinfection, Checking) is initiated.

Sharing among queries with **shared middle subexpressions** can be similarly achieved. Again, such cached results may need to be joined with other events that exist in the respective query to form final results.

B. Advanced Sub-expression Sharing with Different Negative Components

Beyond prior work [1], [2], [3], we now also tackle the case of sharing event expressions when subpatterns contain the same projected positive event types while their negative event types may differ. Besides saving CPU resources, we achieve the added benefit that one sequence result may satisfy several such expressions. If we construct the results for such normalized event expressions of a nested query separately, we may inadvertently produce duplicate results namely one for each of these different event expressions. This then would not only waste CPU resources for re-computation but also incurs the costs associated with duplication removal.

We observe that such event expressions with common positive event types return the same results yet only apply different negation filters. The main idea is that we record the constraints of non-occurrence and non-projected occurrence for each expression at compile time. At run time, as we construct each sequence result, we keep track of which of the given constraints are satisfied (or, rather violated). We stop the evaluation early for unsatisfied event expressions.

Expression-vs-Negative Map (EMap). To facilitate the advanced sequence result generation, we design a data structure *EMap* that records the negative components and non-projected positive components of an expression. Columns in the map correspond to distinct negative components in the shared expressions while rows list the expression identifiers. At compile

[3]In the query plan expressed by a nested AND/SEQ expression, we call the bottommost event expressions *leaf components*.

time, a cell entry indicated by its row and column Map[i, j] is assigned a "1" if the negative event type as indicated by column j is listed in an expression E_i and a "0" otherwise. Possibly one negative component may exist in more than one location in different queries.

SequenceCompute Algorithm: output sequence results

```
Boolean $out \leftarrow true$;
while ($out \wedge stackIndex$ != $0$) do
    Sequence s = Connect(SConstruction(), s); // Recursively call sequence construction until the first stack is reached.
    RVI rvi = BitMarking(); // Mark jth cell "1" if RVI(j) holds true.
    out = SequenceValidation(rvi); // Check filled result vector with EMap.
    stackIndex –;
end while
```

Fig. 6. Sequence Compute with Run-Time Bit Marking

Result Vector Indicator (RVI). In addition, we introduce the *Result Vector Indicator (RVI)* data structure. During query execution, for each partial sequence result we maintain a result vector indicator to check if the current partial result is indeed a correct match. The columns are the same as the ones in EMap. However, we mark the column corresponding to a negative component as "1" if at run time the negative component assigned with that column evaluates to true (not found).

Lemma 1: We stop query evaluation early for one sub-expression E_i if logical AND-ing the bit vectors of the row for E_i in *EMap* with the *RVI* for the partial result is "0".

Lemma 2: We will output a sequence result for a group of shared expressions S if and only if $\exists$ E_i in S for which the logical bit by logical AND-ing the bit vectors of the row for the sub-expression E_i with the current result's *RVI* is "1". Each sequence result is only outputted once for a group of shared expressions. It implies that all the non-existence constraints in at least one of the clustered expressions are satisfied.

Lemma 3: No duplicate results will be produced because we conduct sequence construction only once for all expressions in a group.

The pseudo-code for the shared logic bit-marking based sequence construction strategy is presented in Figure 6. Given flattened event expressions (query disjuncts/conjuncts/leaf components) with the same positive components and one or more different negative components, *EMap* is first constructed. Then, we conduct the sequence construction process for every event instance e_j of the accepting state in the rightmost stack, traversing back along the event pointers. During sequence construction, we also maintain a *RVI* to conduct the sequence validation process. We compare the *RVI* of each partial result with each row of *EMap* continuously. We stop or continue the sequence construction for each partial result based on Lemmas 1 and 2.

Example 11: The normalization procedure rewrites Q_1 = SEQ(Recycle, Washing, ! SEQ(Sharpening, Disinfection, Checking), Operating) into the expression in Figure 7. Figure 8(a) shows the shared instance stacks for all three expressions. Figures 8(b) and 8(c) show the $EMap$ and RVI structures respectively. The negative component for E_1 is ! Checking, for E_2 (! Disinfection, Checking) (Checking is not a positive component as it is not listed in the projection list) and for E_3 (! Sharpening, Disinfection, Checking). When event instance o_{20} of type Operating arrives, the sequence construction is initiated. When evaluating the partial result $< w_5, o_{20} >$, we mark the cell "1" under (! S, D, C) in RVI as $< d_6, c_{16} >$ exists between w_5 and o_{20} and no Sharpening events s_i with 5 < i < 6 exist. Similarly, the (! D, C) AND (! C) cells are marked with "0". The partial result $< w_5, o_{20} >$ can continue the result construction for E_3 because the AND of the bits in the result vector RVI in Figure 8 (c) with the row for E_3 in the $EMAP$ in Figure 8 (b) is "1". Result computation for E_1 and E_2 stopped early by Lemma 1 because the AND of such bits is "0".

SEQ(Recycle, Washing, ! Checking, Operating) OR
Proj$_{R, W, O}$ SEQ(Recycle, Washing, ! Disinfection, Checking, Operating) OR
Proj$_{R, W, O}$SEQ(Recycle, Washing, ! Sharpening, Disinfection, Checking, Operating)

Fig. 7. Normalized Expression for Q1

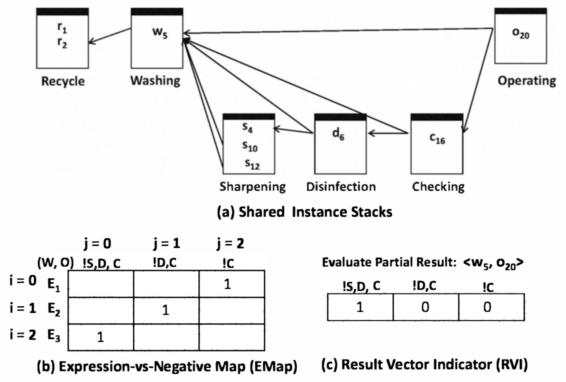


Fig. 8. Bit-Marking Example

V. Plan-Finder

When a set of normalized CEP expressions S share the same positive components, several options arise for grouping them to obtain better shared execution plans. Consider for example the normalized expression S = SEQ(A, B, D) OR SEQ(A, B, ! C, D) OR $Proj_(A, B, D)$SEQ(A, B, ! E, C, D) OR SEQ(A, B, D, E, F) OR SEQ(A, B, D, E, G). The first three conjuncts share the same positive pattern SEQ(A, B, D). The bit-marking algorithm in Section IV-B could be applied to them. Or, alternatively, the first and the last two conjuncts also share the common prefix SEQ(A, B, D). Prefix caching as in Section IV-A could be applied to them. We must make a good choice among these options in the plan space.

A. Problem Definition of Finding Shared-Plans

Given a set of normalized CEP expressions S, we aim to find an expression partition $P = \{g_1, g_2, ..., g_i\}$ with the minimum execution cost among all possible partitions P_i satisfying the following constraints:

- Full coverage: $\forall$ expression E_j in S, $\exists$ g_i that $E_j \in g_i$;
- Non-overlapping: $\forall$ g_i, g_j, $g_i \cap g_j = \emptyset$;
- P_i maps to one execution plan. Each group g_i is mapped to a shared physical operator in Section IV.

Based on our cost analysis for nested and flattened execution plans [19], the Plan-Finder constructs an optimized execution strategy for the normalized form as by Definitions 5, 6 and 7 selected among possible alternatives for estimating the computation sharing.

B. Plan-Finder Search Space

We now study how many possible partitions the Plan-Finder would have to enumerate through to find the best one. To find an optimal solution requires us to enumerate all possible expression partitions. The *Bell number* [21], or the number of different *partitions* P_i of a set S of n elements, describes the size of such a search space, i.e., the total number of all possible partitions for a set of expressions. The problem is challenging, as the complexity of the Plan-Finder is thus exponential.

C. Plan-Finder Search Algorithms

Due to the prohibitive exponential complexity of the search space, we adopt a cost-based heuristic for finding a good quality solution in reasonable time without enumerating the entire search space. While many heuristics are possible, below we sketch one using an iterative refinement methodology:

Selecting a Start Solution. We adopt the strategy to maximally group all event subexpressions with the same positive components into one group to achieve aggressive sharing; though other start heuristics are possible.

Search Strategy: We adopt the iterative improvement method due to its simplicity (see pseudocode in Figure 9). A single basic transformation (e.g., a split of a subset, or merge of two subsets) would transition from a partition solution P_i to its neighbor P_j, e.g., "E_1E_2/E_3E_4" $\rightarrow$ "$E_1E_2/E_3/E_4$".

Selecting a Stop Condition: In general, the search may stop when either k iterations have gone by, or the solution did not improve in the last several rounds, i.e., the search process reaches a plateau. Alternatively, the search can be bounded by resources such as time.

VI. Performance Evaluation

The primary objective of our experimental evaluation is to study the accumulative CPU processing time of the traditional iterative nested execution [11] and our proposed optimized *NEEL* execution strategy with different workloads.

Plan-Finder Algorithm: output best plan

```
1:  partition ← start solution; best-partition ← start solution;
2:  while (not stop condition) do
3:    while (not local_minimum(partition)) do
4:      partition' ← find random solution in NEIGHBORS(partition)
5:      if (cost(partition') < cost(partition)) then
6:        partition ← partition'
7:      end if
8:    end while
9:    if (partition.cost < cost(best-partition)) then
10:     best-partition ← partition
11:   end if
12: end while
13: return best-partition;
```

Fig. 9. Plan-Finder Algorithm

A. Experimental Setup

We have implemented all strategies within the HP stream management system CHAOS [22] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 4GB RAM. We evaluated our techniques using the real stock trades data from [23]. The data contained stock ticker, timestamp and price information. The portion of the trace we used contained 10,000 unique event instances. We used sliding windows with a size of 10ms. In our experiments, the y axis denotes the CPU processing time. CPU processing time means the wall clock time for processing an item e_i in stock trades measured by ($T_{end.ei}$ - $T_{start.ei}$) where $T_{start.ei}$ represents the system time when our processing engine starts processing the data item e_i and $T_{end.ei}$ represents the system time when the engine finishes processing the data item e_i. It is an atomic process, i.e., our processing engine won't stop processing that tuple until it is fully processed.

B. Experimental Design Query Plans

We first evaluate queries by varying three parameter settings including children numbers, query lengths and nesting levels. In Figures 10, 11 and 12, the number of sub-queries is increased from 1 to 3. In Figures 13, 14 and 15, we keep the sub-query number as 1 and increase the sub-query length from 2 to 4. In Figures 16, 17 and 18 we keep the number and the length of sub-queries the same and instead we change the sub-query nesting levels from 1 to 3. Lastly, we evaluate our system with one complex mixed workload in Figure 19.

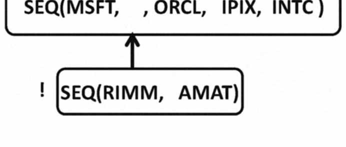
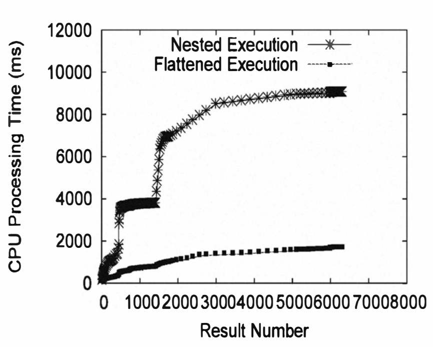
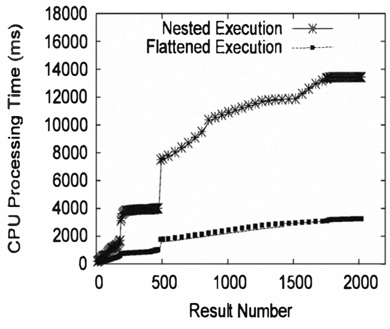


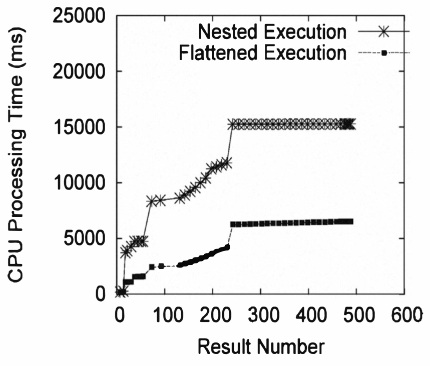
Fig. 10. Sample Query with 1 Child



(a) 1 child as in Figure 10 query

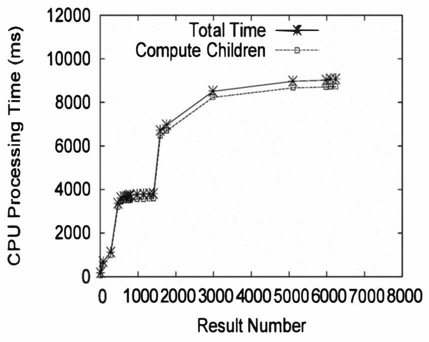


(b) 2 children as in Figure 11 query

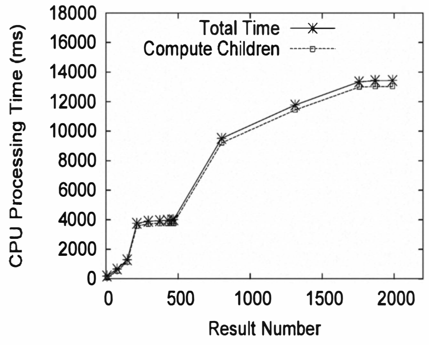


(c) 3 children as in Figure 12 query

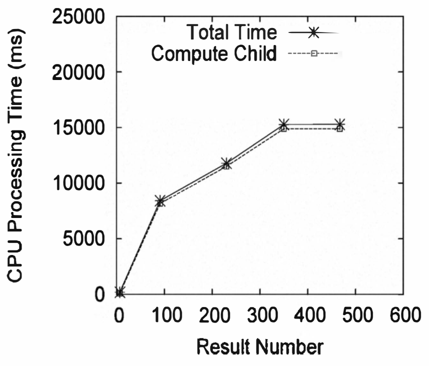
Fig. 20. Varying the Number of Children Queries (as for queries in Figures 10, 11 and 12)



(a) 1 child as in Figure 10 query

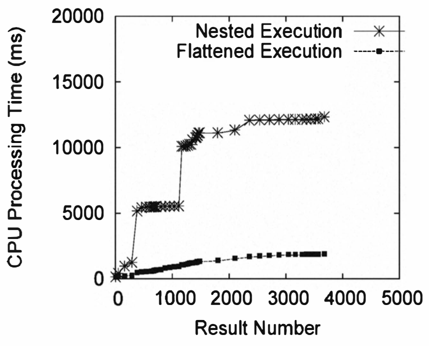


(b) 2 children as in Figure 11 query

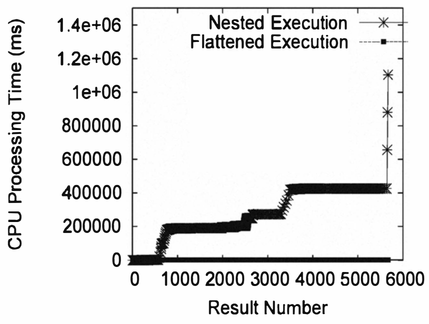


(c) 3 children as in Figure 12 query

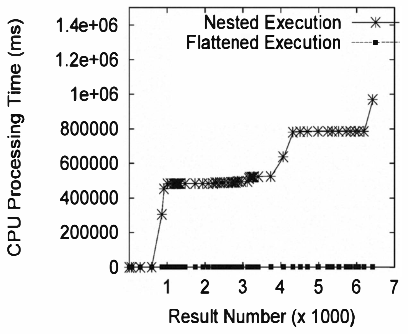
Fig. 21. Comparing Total Computation Time vs. Children Computation Time in the Nest Execution with Increased Children Number



(a) Length 2 as in Figure 13 query



(b) Length 3 as in Figure 14 query



(c) Length 4 as in Figure 15 query

Fig. 24. Varying the Length of Children Queries (as for queries in Figures 13, 14 and 15)

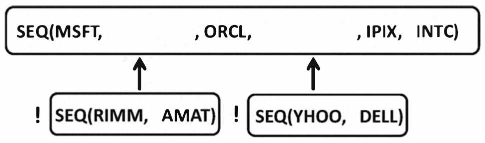


Fig. 11. Sample Query with 2 Children

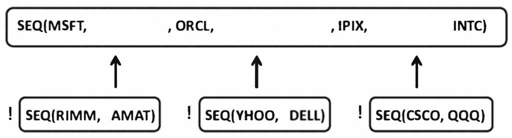


Fig. 12. Sample Query with 3 Children

C. Varying the Number of Children Queries

The first experiment studies queries with increasing numbers of sub-queries as depicted in Figures 10, 11 and 12. In Figure 20, we observe that our proposed optimized *NEEL* execution (Flat) runs on average 5 fold faster than the iterative

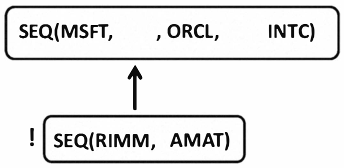


Fig. 13. Sample Query with a Subquery of Length 2

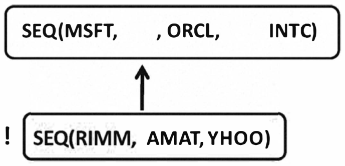


Fig. 14. Sample Query with a Subquery of Length 3

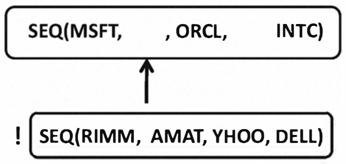


Fig. 15. Sample Query with a Subquery of Length 4

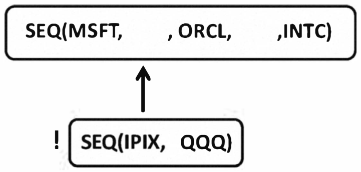


Fig. 16. Sample Query with 2 Nesting Levels

nested execution (Nest). In the Flat execution, we don't need to compute results for SEQ($RIMM$, $AMAT$), SEQ($YHOO$, $DELL$) and SEQ($CSCO$, QQQ). In Figure 21, we observe that in the nested execution, most of the time is used for computing children query results because for each outer partial result, we need to compute children results.

Next, we compare the CPU processing times among the queries in Figures 10, 11 and 12. In Figures 22 and 23, we observe that the query with 3 children generates the least number of results for both nested and flattened execution. The reason for this is that it has more constraints and thus more outer SEQ($MSFT$, $ORCL$, $IPIX$, $INTC$) results can be filtered. In addition, the query with 3 children uses the most CPU processing time among the three queries because of processing more sub-queries. This consumes more CPU processing time. These results match our expectation as clearly the computation time increases with the number of sub-queries and also the probability of finding patterns decreases with an increasing number of event types in the query, i.e., stricter query constraints.

D. Varying the Length of Children Queries

This second experiment processes the queries depicted in Figures 13, 14 and 15 with sub-query lengths varying from 2 to 4. Results are shown in Figure 24. We observe that our proposed optimized *NEEL* execution runs on average several hundred times faster than the iterative nested execution (Nest). In the flattened execution, we don't need to construct all the children query results for SEQ($RIMM$, $AMAT$), SEQ($RIMM$, $AMAT$, $YHOO$) and SEQ($RIMM$, $AMAT$, $YHOO$, $DELL$).

Next, we compare the CPU processing time among queries in Figures 13, 14 and 15 with results shown in Figures 26 and 25. The subquery with length 4 generates the most number of results. As expected, it has less outer SEQ(MSFT, ORCL, INTC) results filtered as the existence of a longer pattern is less likely as compared to the other queries with shorter patterns. In addition, it uses the most CPU processing time among the three queries because it includes the sub-query with the longest length which consumes more computational processing resources.

E. Varying the Nesting Levels of Children Queries

The third experiment processes queries with varying sub-query nesting levels (Figures 16, 17 and 18). Due to space constraints, we omit experimental charts and only report our findings. Our proposed optimized *NEEL* execution again consistently takes less time as compared to nested query execution. It is because the flattened execution doesn't need to construct all the children query results for SEQ($IPIX$, QQQ), SEQ($RIMM$, $AMAT$) and SEQ($YHOO$, $DELL$). Advanced sub-expression sharing with different negative components is applied during query evaluation. Thus significant CPU processing resources are saved. In addition, the query with the largest nesting levels generates the most number of results and uses the most CPU processing time among the three queries for both nested and flattened execution. One, the query includes the sub-query with the largest nesting levels which consumes more time to be computed. Two, in the nested execution, less outer SEQ($MSFT$, $ORCL$, $INTC$) results are filtered as it is relatively infrequent to have events of more nesting levels occur in a sequence.

F. Mixed Workload

The last experiment processes the mixed complex query in Figure 19. Our rewriter transforms it into the normalized expression $E = E_1$ (SEQ(MSFT, ! IPIX, ORCL, INTC)) OR E_2 (SEQ(MSFT, ! QQQ, ORCL, INTC)) OR E_3 (SEQ(MSFT, ! RIMM, ORCL, INTC)) OR E_4 (SEQ(RIMM, DELL, AMAT, MSFT, ORCL)) OR E_5 (SEQ(IPIX, DELL, AMAT, MSFT, ORCL)) OR E_6 ($Proj_{CSCO,YHOO,QQQ}$SEQ(CSCO, ! RIMM, YHOO, QQQ)) OR E_7 ($Proj_{CSCO,YHOO,QQQ}$SEQ(CSCO, ! IPIX, RIMM, YHOO, QQQ)). The partition returned by the plan-Finder has three groups: $\{[E_1, E_2, E_3], [E_4, E_5], [E_6, E_7]\}$. The group $[E_1, E_2, E_3]$ is mapped to the operator in Section IV-B as these subexpressions share the same positive event types (MSFT, ORCL, INTC) while the negative event

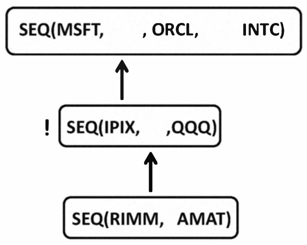


Fig. 17. Sample Query with 3 Nesting Levels

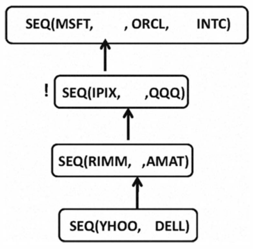


Fig. 18. Sample Query with 4 Nesting Levels

types are different. Similarly, $[E_6, E_7]$ is also mapped to the operator in Section IV-B. $[E_4, E_5]$ is mapped to the operator in Section IV-A as they share the same suffix (DELL, AMAT, MSFT, ORCL). As expected, our proposed *NEEL* execution takes significantly less time as compared to the iterative nested execution as shown in Figure 27.

VII. Related Work

To the best of our knowledge, existing CEP systems [1], [2], [3], [9], [24], [15] mostly support the execution of only flat sequence queries. While CEDR [9] allows applying negation over composite event types within their proposed language, the execution strategy for such nested queries is not discussed. In addition, no work has been reported on tackling the performance deficiency when applying negation over composite event types. ZStream [3] considers the ordering of execution for CEP queries using a tree-based query plan – similar to join ordering in traditional relational databases. It only supports negation over primitive event types. ZStream doesn't consider optimization over multiple expressions nor of nested CEP expressions. SASE [1], [17] considers flat queries and negation is applied as a final filtration step. Cayuga [2] only allows sub-queries in the FROM clause and it also doesn't support applying negation over composite event types. In short, no processing mechanism for CEP queries with nested complex negation has been proposed in the literature to date.

Complex pattern queries often contain common or similar sub-expressions within a single query or also among multiple distinct queries. Multiple-query optimization in databases [25], [26], [27] typically focussed on static relational databases, identifies common subexpressions among queries such as common joins or filters. Multiple expression sharing for stack-based pattern evaluation for CEP queries has not yet been studied. In particular, our work is the first to share the processing of CEP expressions with the same positive event types interleaved with different negative event types.

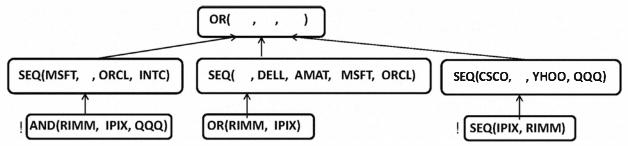


Fig. 19. Mixed Workload

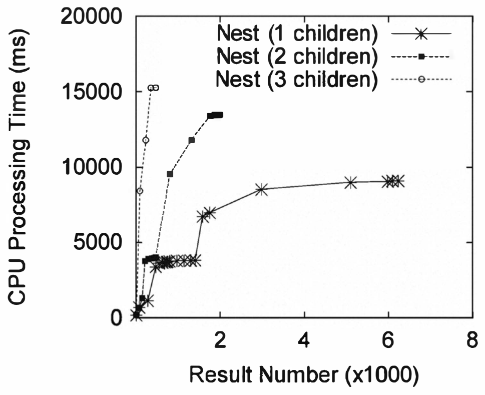


Fig. 22. Nested Execution with Increased Children Number as in Figures 10, 11 and 12 queries

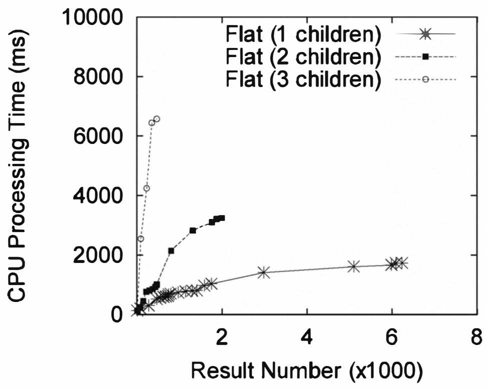


Fig. 23. Flattened Execution with Increased Children Number as in Figures 10, 11 and 12 queries

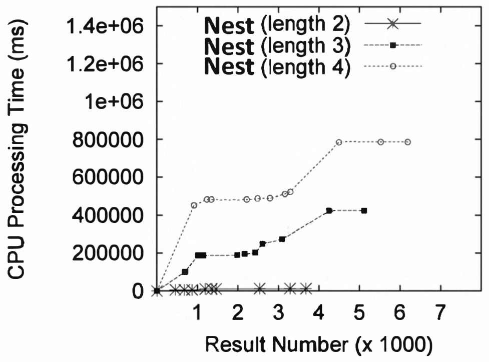


Fig. 25. Varying the Length of Children Queries as in Figures 13, 14 and 15 queries

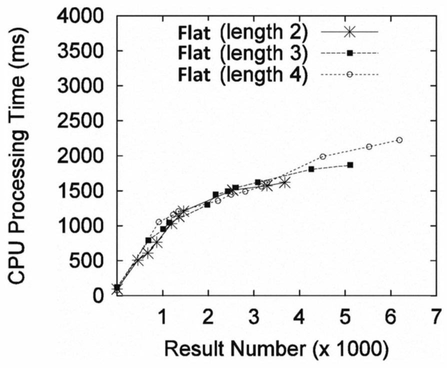


Fig. 26. Varying the Length of Children Queries as in Figures 13, 14 and 15 queries

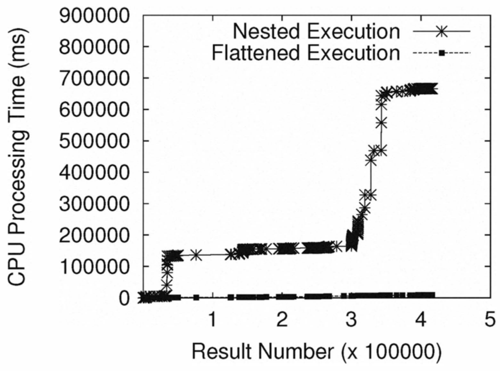


Fig. 27. Mixed Workload as in Figure 19 query

VIII. Conclusion

This paper describes the first work on comprehensively supporting nested query specification and execution in the CEP context. The CEP query language *NEEL* allows users to specify fairly complex queries in a compact manner with both temporal relationships and negation well-supported. A query plan for the execution of nested CEP queries is designed. This nested query plan model permits a direct implementation of nested CEP queries following the principle of nested query execution for SQL queries. However, such direct query execution suffers from several performance deficiencies. We thus design a normalization procedure converting a nested event expression into a normal form. We propose prefix caching, suffix clustering and a customized "bit-marking" physical execution strategy that efficiently process a group of similar subexpressions. An optimizer that employs iterative improvement capturing the optimal shared execution method is also designed. As demonstrated by our experiments, in many cases our optimized *NEEL* execution performs 100 fold faster than the traditional iterative nested execution.

Acknowledgements. This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF 1018443 and NSF IIS 0917017, Turkish National Science Foundation TUBITAK under career award 109E194.

References

[1] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams." in *SIGMOD Conference*, 2006, pp. 407–418.
[2] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system." in *CIDR*, 2007, pp. 412–422.
[3] Y. Mei and S. Madden, "Zstream: a cost-based query processor for adaptively detecting composite events," in *SIGMOD*, 2009, pp. 193–206.
[4] J. M. Boyce and D. Pittet, "Guideline for hand hygiene in healthcare settings," *MMWR Recomm Rep.*, vol. 51, pp. 1–45, 2002.
[5] V. Shnayder, B. rong Chen, K. Lorincz, T. R. F. F. Jones, and M. Welsh, "Sensor networks for medical care," in *SenSys*, 2005, p. 314.
[6] D. I. Tapia, J. A. Fraile, S. Rodrłguez, J. F. de Paz, and J. Bajo, "Wireless sensor networks in home care," in *IWANN (1)*, 2009, pp. 1106–1112.
[7] W. Kim, "On optimizing an sql-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, 1982.
[8] N. May, S. Helmer, and G. Moerkotte, "Nested queries and quantifiers in an ordered context," in *ICDE*, 2004, pp. 239–250.
[9] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing." in *CIDR*, 2007, pp. 363–374.
[10] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE*, 1996, pp. 450–458.
[11] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "Processing nested complex sequence pattern queries over event streams," in *DMSN*, 2010, pp. 14–19.
[12] M. Brantner, C.-C. Kanne, G. Moerkotte, and S. Helmer, "Algebraic optimization of nested xpath expressions," in *ICDE*, 2006, p. 128.
[13] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite events for active databases: Semantics, contexts and detection." in *VLDB*, 1994, pp. 606–617.
[14] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "NEEL: The nested complex event language for real-time event analytics," *in BIRTE*, 2010.
[15] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool, "Sequence pattern query processing over out-of-order event streams," in *ICDE*, 2009, pp. 784–795.
[16] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *SIGMOD Conference*, 2008, pp. 147–160.
[17] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting kleene closure over event streams," in *ICDE*, 2008, pp. 1391–1393.
[18] J. M. Smith and P. Y.-T. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. ACM*, vol. 18, no. 10, pp. 568–579, 1975.
[19] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta, "Nested complex sequence pattern query processing over event streams: Rewriting and compacting," Worcester Polytechnic Institute, Technical Report WPI-CS-TR-10-15, 2010.
[20] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal, "AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering," in *VLDB*, 2006, pp. 559–570.
[21] M. Klazar, "Bell numbers, their relatives, and algebraic differential equations," *J. Comb. Theory, Ser. A*, pp. 63–87, 2003.
[22] C. Gupta, S. Wang, I. Ari, M. C. Hao, U. Dayal, A. Mehta, M. Marwah, and R. K. Sharma, "Chaos: A data stream analysis architecture for enterprise applications," in *CEC*, 2009, pp. 33–40.
[23] "I. inetats. stock trade traces. http://www.inetats.com/."
[24] M. Liu, E. A. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta, "E-cube: Multi-dimensional event sequence processing using concept and pattern hierarchies," in *ICDE*, 2010, pp. 1097–1100.
[25] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, pp. 23–52, 1988.
[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD*, 2000, pp. 249–260.
[27] Finkelstein and Sheldon, "Common expression analysis in database applications," in *SIGMOD*, 1982, pp. 235–245.