Plan-based Complex Event Detection across Distributed Sources

Mert Akdere, Ugur Cetintemel and Nesime Tatbul

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-08-01
February 2008

Plan-based Complex Event Detection across Distributed Sources

Mert Akdere, Uğur Çetintemel
Brown University
makdere,ugur@cs.brown.edu

Nesime Tatbul
ETH Zurich
tatbul@inf.ethz.ch

ABSTRACT

Complex Event Detection (CED) is emerging as a key capability for many monitoring applications such as intrusion detection, sensor-based activity and phenomena tracking, and network monitoring. Existing CED solutions commonly assume centralized availability and processing of all relevant events, and thus incur significant overhead in distributed settings. In this paper, we present and evaluate communication-efficient techniques that can efficiently perform CED across distributed event sources.

Our techniques are *plan-based*: we generate multi-step event acquisition and processing plans that leverage temporal relationships among events and event occurrence statistics to minimize event transmission costs while meeting application-specific latency expectations. We present an optimal but exponential-time dynamic programming algorithm and two polynomial-time heuristic algorithms, as well as their extensions for detecting multiple complex events with common sub-expressions. We characterize the behavior and performance of our solutions via extensive experimentation on synthetic and real-world data sets.

1. INTRODUCTION

In this paper, we study the problem of complex event detection (CED) in a monitoring environment that consists of potentially a large number of distributed event sources (e.g., hardware sensors or software receptors). CED is becoming a fundamental capability in many domains including network and infrastructure security (e.g., denial of service attacks and intrusion detection [20]), phenomenon and activity tracking (e.g., fire detection, storm detection, tracking suspicious behavior [21]). More often than not, such sophisticated (or "complex") events "happen" over a period of time and region. Thus, CED often requires consolidating over time "simple" events generated by distributed sources.

Existing CED approaches, such as those employed by stream processing systems [15, 16], triggers [1], and active databases [8], are based on a centralized, push-based event acquisition and processing model. Sources generate simple events, which are continually pushed to a processing site where the registered complex events are evaluated in the form of continuous queries, triggers, or rules. This model of CED is neither efficient, as it requires communicating all base events to the processing site, nor necessary, as only a fraction of all base events eventually make up complex events.

This paper presents a new plan-based approach for communication-efficient CED across distributed sources. Given a complex event, we generate a cost-based multi-step detection plan on the basis of the temporal constraints among constituent events and event frequency statistics. Each step in the plan involves acquisition and processing of a different subset of the events with the basic goal of postponing the monitoring of high frequency events to later steps in the plan. As such, processing the higher frequency events conditional upon the occurrence of lower frequency ones eliminates the need to communicate the former in many cases, thus has the potential to yield substantial savings in transmission costs.

Our algorithms are parameterized to limit event detection latencies by constraining the number of steps in a CED plan. There are two uses for this flexibility: First, the local storage available at each source dictates how long events can be stored locally and would thus be available for retrospective acquisition. In other words, we can limit the execution duration of our plans to respect *event lifetimes* at sources. Second, while timely detection of events is critical in general, it is also the case that some applications are more delay-tolerant than others (e.g., especially non-automated, human-in-the-loop applications), allowing us to generate more efficient plans.

To implement this approach, we first present a dynamic programming algorithm that is optimal but runs in exponential time. We then present two polynomial-time heuristic algorithms. In both cases, we discuss a practical but effective approximation scheme that limits the number of candidate plans considered to further trade off the plan quality and cost. An integral part of planning is cost estimation, which requires effective modeling of event behavior. We show how commonly used distributions and histograms can be used to model events with independent and identical distributions and then extend the models to support temporal dependencies such as burstiness. We also study CED in the presence of multiple complex events and describe extensions that leverage shared sub-expressions for improved performance. Finally, we quantify the behavior and benefits of our algorithms and extensions on a variety of workloads, using synthetic and real-world data (obtained from PlanetLab).

The rest of the paper is structured as follows. An overview of our event detection framework is provided in Section 2. Our plan-based approach to CED with plan generation and execution algorithms is described in Section 3. In Section 4, we discuss the details of our cost and latency models. Section 5 extends plan optimization to shared subevents and event constraints. We present our experimental results in Section 6, cover the related work in Section 7, and conclude with future directions in Section 8.

2. BASIC FRAMEWORK

Events are defined as activities of interest in a system [10]. Detection of a person in a room, the firing of a cpu timer, and a Denial of Service (DoS) attack in a network are example events from various application domains. All events signify certain activities, however their complexity degrees can be significantly different. For instance, the firing of a timer is instantaneous and simple to detect, whereas detection of DoS is an involved process that requires computation over many simpler events. Correspondingly, events are categorized as *primitive* (a.k.a. base) and *complex* (a.k.a. compound), basically forming a hierarchy of events, where complex events are generated by composing primitive or other complex events using a set of event composition operators (Section 2.2).

We use *interval-based semantics*: each event has an associated time-interval that indicates its occurrence period. For primitive events, this interval is a single point (i.e., identical start and end points) when the event occurs. For complex events, the assigned intervals contain the time intervals of *all* subevents. Interval-based semantics better capture the underlying event structure and does not suffer from some well-known correctness problems that arise with point-based semantics [9].

2.1 Primitive Events

Each event type (primitive or complex) has a schema that extends the base schema consisting of the following required attributes:

- **node_id** is the id of the node that generated the event.
- **event_id** is a unique id for each event instance. It can be made unique for every event instance or can be set to a function of event attributes for *similar* event instances to get the same id. For example, in an RFID enabled library application a book might be detected by multiple RFID receivers at the same time. Such readings can be discarded if they are assigned the same event identifier.
- **start_time** and **end_time** represent the time interval of the event and are assigned by the system based on the event operator semantics explained in the next subsection. These time values come from an ordered domain.

Finally, a reserved but nonmandatory attribute, named *maxlatency*, is used to specify the latency deadline for an event type. When a latency deadline is specified, the system will only consider the plans satisfying the latency requirement.

Primitive event declarations specify the details of the transformation from raw source data into primitive events. The syntax for primitive event declaration is:

primitive `name`
on `source_list`
schema `attribute_list`

Each primitive event is assigned a unique name using `name`. The set of sources used in a primitive event is listed in the `source_list`. The *schema* component expresses the names and domains of the attributes of the primitive event type and automatically inherits the attributes in the base schema.

An example primitive event, expressing the detection of a person, is shown below together with the declaration of a *person_detector* source (e.g., a face detection module running on a smart camera).

source person_detector
schema int id, double loc_x, double loc_y

primitive person_detected
on person_detector as PD, node
schema event_id **as** hash_f(person_detected, node.id, node.time, PD.id),
loc **as** [PD.loc_x, PD.loc_y],
person_id **as** PD.id

We use the pseudo-source `node` which enables access to context information such as the location of the source and the current value of node clock. We use a hash function, `hash_f`, to generate unique ids for event instances. Similar to its use in SQL, **as** describes how an attribute is derived from others.

2.2 Event Composition

Complex events are specified on simpler events using the syntax:

complex `name`
on `source_list`
schema `attribute_list`
where `constraint_list`

A unique name is given to each complex event type using the `name` attribute. Subevents of a complex event type, which can be other complex or primitive events, are listed in `source_list`. As in primitive events, the source list may contain the `node` pseudo-source as well. The `attribute_list` contains the attributes of a complex event type which together form a super set of the base schema and also describes the way they are assigned values. In this sense, the schema specifies the transformation from subevents to complex events.

In most applications, users will be interested in complex events which impose constraints on their subevents. For instance, users may want to monitor events occurring in nearby locations or same time intervals. In order to support such constraints, our system allows temporal, spatial, attribute-based and existential constraints to be specified in the **where** clause of a complex event specification.

We use a standard set of event composition operators for easy specification of causal and temporal correlations between subevents which could otherwise be expressed as a set of attribute constraints on start and end times. Our event operators, `and`, `or` and `seq`, are all *n-ary* operators. In addition, we allow non-existence constraints to be expressed through the use of the negation operator `!`. We also extended the event operators with time windows for temporal constraint specification. The time window argument, w, of an event operator specifies the maximum time duration between the occurrence of any two subevents of a complex event instance. Hence, all the subevents are separated by at most w time units.

Formal semantics of our operators are provided below, where we denote subevents with $e_1, e_2, \ldots, e_n$ and the start and end times of the output complex event with t_1 and t_2.

- **and**$(e_1, e_2, \ldots, e_n; w)$ outputs a complex event with $t_1 = min_{i\in\{1,..,n\}}\, e_i.\,start\_time, t_2 = max_{i\in\{1,..,n\}} e_i.\,end\_time$ if $max_{i,j\in\{1,..,n\}} e_i.\,end\_time - e_j.end\_time <= w$. Note that the subevents can happen in any order.
- **seq**$(e_1, e_2, \ldots, e_n; w)$ outputs a complex event with $t_1 = e_1.start\_time$, $t_2 = e_n.end\_time$ if (i) $e_i.end\_time < e_{i+1}.start\_time$ for $i = 1, \ldots, n-1$, (ii) $e_n.end\_time - e_1.end\_time \leq w$. Hence, `seq` is a restricted form of `and` where overlapping is not allowed and events need to occur in order.
- **or**$(e_1, e_2, \ldots, e_n)$ outputs a complex event when a subevent occurs. t_1 and t_2 are set to start and end times of the subevent. Note that this operator does not require a window argument.
- **negation**, denoted by **!**, can be applied to the events inside `and` and `seq` operators to express non-occurrence constraints. For the event `and(`$e_1, e_2, .., !e_i, .., e_n; w$`)` we need $\nexists\, e_i : min_j\,(e_j.end\_time) \leq e_i.end\_time \leq max_j\,(e_j.end\_time)$ over all non-negated subevents e_j. For `seq(`$e_1, e_2, .., !e_i,$ $.., e_n; w$`)` we need to have $\nexists\, e_i : e_p.end\_time \leq e_i.end\_time \leq e_q.start\_time$ where e_p and e_q are the previous and next non-negated subevents for e_i. At least one of the subevents in a complex event should be left non-negated.

Parameterized attribute-based constraints between events and value-based comparison constraints can be specified in the where clause as well. We illustrate the use of the constraints through the running person event below:

complex running_person
on person_detected **as** PD1,person_detected **as** PD2, node
schema event_id **as** hash_f(running_person, node.id, node.time, person_id),
loc **as** PD2.loc,
person_id **as** PD1.person_id
where **seq**(PD1,PD2;3) and PD1.person_id = PD2.person_id
and distance(PD1.loc, PD2.loc) $\geq$ 12

2.3 Event Detection Graphs

Our event detection model is based on *event detection graphs* [8] constructed from the user given event specifications expressed in our language. For every event expression we construct an event detection tree and these event trees are then merged to form the event detection graph. Common events in different event trees, the shared events, are merged to form nodes with multiple parents. Nodes in an event detection graph are either operator nodes or primitive event nodes. The non-leaf nodes, which execute the event language operators on their inputs are the operator nodes. The inputs to operator nodes are events (either complex or primitive) coming from their child nodes and their outputs are complex events. The leaf nodes in the graph are primitive event nodes. A primitive event node exists for each primitive event type and stores references to the instances of that primitive event type.

2.4 System Architecture

The main components in our system are the event *sources* and the *base* node (Figure 1). Sources generate events; e.g., routers and firewalls in a network monitoring application, and a temperature sensor in a disaster monitoring application are examples. Sources have local storage that allow them to log events of interest temporarily. These logs can be queried and events be acquired when necessary. In practice, some event sources may not have any local storage or be autonomous and outside our control (e.g., RSS sources on the web). In such cases, we rely on *proxy* nodes that provide these capabilities on their behalf. Thus, we use source when referring to either the original event source or its proxy.

The base station is responsible for generating and executing CED plans. Plan execution involves coordination with event sources (or their proxies as mentioned above) as event types are transmitted upon demand from the base. Consequently, our system combines the pull and push paradigms of data collection to avoid the disadvantages of a purely push-based system. The CED plans we generate strive to reduce the network traffic towards base station by carefully choosing which sources will transmit data.

3. PLANNING FOR CED

3.1 Event Detection Plans: Overview

A common approach to event detection would be to continuously transmit all the events to the base where they would be processed as soon as possible. This primarily push-based approach is typical of continuous query processing systems (e.g., [15, 16, 17]). From an efficiency point of view, this approach leads to a hot-spot at the base and significant resource consumption at event sources for event transmission. From a semantic point of view, many applications do not require access to all "raw" events but only a small fraction of relevant ones. Our goal is to avoid continuous global acquisition of data *without missing any relevant events*, as specified by the users.

To achieve this goal, we use *event detection plans* to guide event acquisition decisions. The simplest plan consists of a single step in which all subevents are simultaneously monitored (referred to as the *naive plan* in the sequel). More complex plans have up to *n* steps, where *n* is the number of subevents, each involving the monitoring of a subset of events. The number of plans for a complex event with *n* primitive subevents is exponential in *n* as given by the recursive relation $T(n) = \sum_{i=1}^{n} \binom{n}{i} T(n-i)$, where we define $T(0)$ to be 1.

We use a *cost-latency model* based on event occurrence probabilities to calculate the expected costs and latencies of candidate event detection plans. We define the *expected cost of a plan* as the expected number of events the plan asks nodes to send to the base per time unit. We expect transmission costs to be the bottleneck for many networked systems, especially for sensor networks with thin, wireless pipes. Even with Internet-based systems, bandwidth problems arise, especially around the base, with increasing event generation rates. Additionally, we define the *latency of a plan* for a complex event as the time between the occurrence of the event and its detection by the system executing the plan. We assume that there is an estimated latency to access each event source and that detection latencies are dominated by network latencies, thus ignoring the event processing costs at the base station. However, since we strive to decrease the number of events sent to base, our approach should reduce *both* network and processing costs. Note that we abstractly define both metrics to avoid overspecializing our results to particular system configurations and protocol implementations.

As briefly mentioned earlier, latency expectations for complex events may originate from two different sources. First, we may have user specified, explicit latency deadlines based on the semantics and requirements of associated applications. Second, latency deadlines can arise from limited data logging capabilities: an event source may be able to store event instances only for a limited time before it runs out of space and has to delete data. Therefore, a plan that assumes the availability of events for longer periods are not going to be useful. In practice we can consider both cases and use the most strict latency target for a complex event.

To illustrate our cost-latency model, consider a simple complex event $\mathsf{and}(e_1, e_2)$. The cost of the naive plan for detecting this event would be the sum of costs of e_1 and e_2. On the other hand, a two-step plan, where we first monitor e_1 and then acquire e_2 if e_1 occurs, could cost less but would incur higher latency than the naive plan, which offers the minimum possible latency. This tradeoff between cost and latency is an important focus of our work: our goal is to find low-cost event detection plans that meet event-specific latency expectations.

Event detection plans specify monitoring orders for the subevents of complex events. We represent our plans with extended finite state machines (FSMs). Consider the complex event $\mathsf{and}(e_1, e_2, e_3; w)$ where e_1, e_2, e_3 are primitive events and *w* is the window size. There are T(3) = 13 different detection plans for this complex event. State machines of the plans for this complex event have at most $n = 3$ states (except the final state), in each of which a subset of primitive events is monitored. One state machine of each size is given in Figure 2. For instance, the 3-step monitoring plan: "First, continuously monitor e_1, then on e_1 lookup e_2, and finally on e_1 and e_2 lookup e_3", is illustrated in Figure 2(c), where the notation $e_1 \rightarrow e_2 \rightarrow e_3$ is used to denote this plan.

The FSMs we use for representing plans are *nondeterministic*, since they can have multiple active states at a time. Every active state corresponds to a partial detection of the complex event. For example, in state S_{e_1} of the plan given in Figure 2(c), there can be active instances of e_1 waiting for instances of e_2. When an instance of e_2 is detected, in addition to the transition to next state, a self-transition will also occur so that an instance of e_1 can match multiple instances of e_2 (self-transitions are not shown in the figure). Unlike the initial state that is always active, intermediate states are active only as long as the windowing constraints among event instances are met.

3.2 Plan Generation

We now describe how plans are generated for a complex event with the goal of optimizing the overall monitoring costs while respecting latency constraints. First, we consider the problem of plan generation for a complex event defined by a single operator. We provide two algorithms for this problem: a dynamic programming solution and a heuristic method (in sections 3.2.1 and 3.2.2, respectively). For a given complex event, both algorithms produce a set of pareto-optimal plans with different cost-latency characteristics. Then, in section 3.2.3, we generalize our approach to more com-

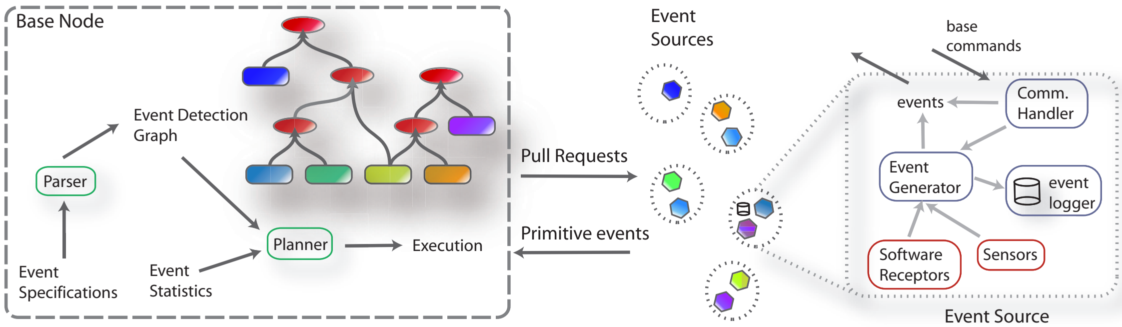


Figure 1: Complex event detection framework: The base node plans and coordinates the event detection using low network cost event detection plans formed by utilizing event statistics. The event detection model is an event detection graph generated from the given event specifications. Information sources feed the system with primitive events and can operate both in pull and push based modes.

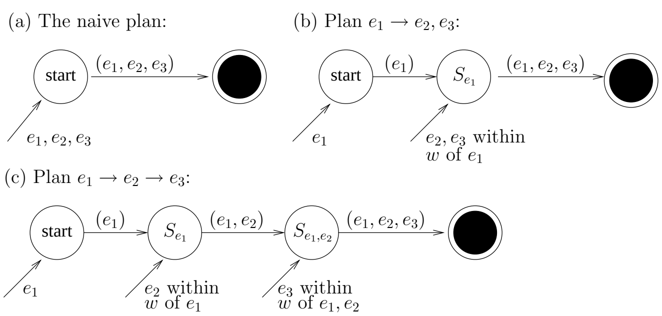


Figure 2: Event detection plans represented as finite state machines

plicated events by describing a hierarchical plan generation method that uses as building blocks the candidate plans generated for simpler events. The dynamic programming algorithm can find optimal plans and achieve the minimum global cost. However, it has exponential time complexity and is thus only applicable to small problem instances. The heuristic algorithm, on the other hand, runs in polynomial time and, while it cannot guarantee optimality, it produces near optimal results for the cases we studied (Section 6).

3.2.1 *The dynamic programming approach*

The input to the dynamic programming plan generation algorithm is a complex event C defined over the subevents S and a set of plans for monitoring each subevent. For the primitive subevents, the only monitoring plan is the single step plan, whereas for the complex subevents there can be multiple monitoring plans. Given these inputs, the dynamic algorithm produces a set of *pareto optimal* plans for monitoring the complex event C. These plans will then be used in the hierarchical plan generation process to produce plans for higher-level events (Section 3.2.3).

A plan is *pareto optimal* if and only if no other plan can be used to reduce cost or latency without increasing the other metric.

Definition 1. A plan p_1 with cost c_1 and latency l_1 is *pareto optimal* if and only if $\nexists\, p_2$ with cost c_2 and latency l_2 such that ($c_1 > c_2$ and $l_1 \geq l_2$) or ($l_1 > l_2$ and $c_1 \geq c_2$).

The dynamic programming solution to plan generation is based on the following *pareto optimal substructure property*: Let $t_i \subseteq S$ be the set of subevents monitored in the i^{th} step of a pareto optimal plan p for monitoring C. Define p_i to be the subplan of p, consisting of its first i steps used for monitoring the subevents $\cup_{j=1}^{i} t_j$. Then the subplan p_{i+1} is simply the plan p_i followed by a single step in which the subevents t_{i+1} are monitored. The pareto optimal substructure property can then be stated as: if p_{i+1} is pareto optimal then p_i must be pareto optimal.

PROOF : PARETO OPTIMAL SUBSTRUCTURE. Let the cost of p_i be c_i and its latency be l_i. Assume that p_i is not pareto optimal. Then by definition $\exists p'_i$ with cost c'_i and latency l'_i such that ($c_i > c'_i$ and $l_i \geq l'_i$) or ($l_i > l'_i$ and $c_i \geq c'_i$). However, then p'_i could be used to form a p'_{i+1} such that ($c_{i+1} > c'_{i+1}$ and $l_{i+1} \geq l'_{i+1}$) or ($l_{i+1} > l'_{i+1}$ and $c_{i+1} \geq c'_{i+1}$) which would contradict the pareto optimality of p_{i+1}. □

This property implies that, if p, the plan used for monitoring the complex event C, is a pareto optimal plan, then p_i for all i, $1 \leq i \leq |S|$, must be pareto optimal as well. Our dynamic programming solution leveraging this observation is shown in Algorithm 1 for the special case where all the subevents are primitive. Generalization of this algorithm to the case with complex subevents (not shown here due to space constraints) basically requires repeating the lines between 6 and 15 for all possible plan configurations of monitoring events in set s in a single step. After execution, all

Algorithm 1 Dynamic programming solution to plan generation

```
1.  Input: S = {e_1, e_2, ..., e_N}
2.  for plength = 1 to |S| do
3.    for all s ⊆ S \ ∅ do
4.      p = new plan
5.      t = S \ s
6.      if plength != 1 then
7.        for all j ⊆ t \∅ do
8.          for all plan p_j in poplans[j] do
9.            p.steps = p_j.steps
10.           p.steps.add(new step(s))
11.           if p is pareto optimal for poplans[s ∪ j] then
12.             poplans[s ∪ j].add(p)
13.     else
14.       p.steps.add(new step(s))
15.       poplans[s].add(p)
```

pareto optimal plans for the complex event C will be in $poplans[S]$, where $poplans$ is the pareto optimal plans table. This table has exactly $2^{|S|}$ entries, one for each subset of S. Every entry stores a list of pareto optimal plans for monitoring the corresponding subset of events. Moreover, the addition of a plan to an entry $poplans[s]$ may render another plan in $poplans[s]$ non-pareto optimal. Hence, when adding a pareto optimal plan to the list (line 12), we remove the non-pareto optimal ones.

At iteration i of the plength for loop, we are generating plans of length (number of steps) i, whose first $i-1$ steps consist of the events in set $j \subseteq t$ and last step consists of the events in set s. Therefore, in the i^{th} iteration of the plength for loop, we only need to consider the sets s and j that satisfy:

$$|t| + 1 \geq i \Rightarrow |t| \geq i - 1 \tag{1}$$

$$\Rightarrow |t| = |S| - |s| \geq i - 1 \Rightarrow |s| \leq |S| - i + 1 \tag{2}$$

$$|j| \geq i - 1 \tag{3}$$

Otherwise, at iteration i, we would redundantly generate the plans with length less than i. However, for simplicity we have not included those constraints in the pseudocode shown in Algorithm 1 as they do not change the correctness of the algorithm.

Finally, the analysis of the algorithm (for the case of primitive subevents) reveals that its complexity is $O(|S|2^{2|S|}k)$, where the constant k is the maximum number of pareto optimal plans a table entry can store. When the number of pareto optimal plans is larger than the value of k: (i) non-pareto optimal plans may be produced by the algorithm, which also means we might not achieve global optimum and; (ii) we need to use a strategy to choose k plans from the set of all pareto optimal plans. To make this selection, we explored a variety of strategies, including naive random selection, selection ranked by cost or latency, and a mixture approach that selects a uniform mix of low cost, low latency, and low cost-latency (a linear combination of the two factors) plans. We discuss these alternatives and experimentally compare them in Section 6.

3.2.2 *Heuristic techniques*

Even for moderately large instances of complex events, enumeration of the plan space for plan generation is not a viable option due to its exponential size. As discussed earlier, the dynamic programming solution requires exponential time as well. To address this tractability issue, we have come up with a strategy that combines the following two heuristics, which together generate a representative subset of all plans with distinct cost and latency characteristics:

- Forward Stepwise Plan Generation: This heuristic starts with the minimum latency plan, a single-step plan with the minimum latency plan selected for each complex subevent, and repeatedly modifies it to generate lower cost plans until the latency constraint is exceeded or no more modifications are possible. At each iteration, the current plan is transformed into a lower cost plan either by moving a subevent detection to a later state or replacing the plan of a complex subevent with a cheaper plan.

- Backward Stepwise Plan Generation: This heuristic starts by finding the minimum cost plan, i.e., an n-step plan with the minimum cost plans selected for each complex subevent, where n is the number of subevents. This can be found in a greedy way when all subevents are primitive, otherwise a nonexact greedy solution which orders the subevents in increasing $cost \times occurrence\ frequency$ order can be used. At each iteration, the plan is repeatedly transformed into a lower latency plan either by moving a subevent to an earlier step or changing the plan of a complex subevent with a lower latency plan, until no more alterations are possible.

Thus, the first heuristic starts with a single-state FSM and grows it (i.e., adds new states) in successive iterations, whereas the second one shrinks the initially n-state FSM (i.e., reduces the number of states). Moreover, both heuristics choose the move with the highest cost-latency gain at each iteration and both finish in a finite number of iterations since the algorithm halts as soon as it cannot find a move that results in a better plan. In other words, while the first heuristic aims to generate low-latency plans with reasonable costs, the latter strives to generate low-cost plans meeting latency requirements.

As a final step, the plans produced by both heuristics are merged into a feasible plan set, one that meets latency requirements. During the merge, only the plans which are pareto optimal within the set of generated plans are kept. As is the case with the dynamic programming algorithm, only a limited number of these plans will be considered by each operator node for use in the hierarchical plan generation algorithm. The selection of this limited subset is performed as discussed in the previous subsection.

3.2.3 *Hierarchical plan composition*

Plan generation for a multi-level complex events proceeds in a hierarchical manner where the plans for the higher level events are built using the plans of the lower level events. The process follows a depth-first traversal on the event detection graph, running a plan generation algorithm at each node visited. The plans produced at a node are propagated to the parent node, which uses them when creating its own plans.

Generating only the minimum latency or the minimum cost plan at each node does not guarantee globally optimal solutions, as the global optimum might include high-cost, low-latency plans for some component events and low-cost, high-latency plans for the others. To address the complexity, each node creates a set of plans with a variety of latency and cost characteristics.

The dynamic programming algorithm produces exclusively pareto optimal plans, which are essential since *non-pareto optimal plans lead to suboptimal global solutions* (see the proof below). Moreover, if the number of pareto optimal plans submitted to parent nodes is not limited, then using the dynamic programming algorithm for each complex event node we can find the global optimum selection of plans. Yet, as mentioned before the size of this pareto optimal subset is limited by a parameter trading computation with the explored plan space size. On the other hand, the set of plans produced by the heuristic solution does not necessarily contain the pareto optimal plans within the plan space. As a result, even when the number of plans submitted to parent nodes is not limited, the heuristic algorithm still does not guarantee optimal solutions.

PROOF : NON-PARETO OPTIMAL PLANS ARE SUBOPTIMAL. Let C be a complex event with s being one of its complex subevents. The case where all the subevents are primitive is trivial since primitive events only have a single monitoring plan that is by definition pareto optimal. Let plan p_c with cost and latency (c_c, l_c) be the optimal plan, a minimum cost plan with respect to a latency threshold, used for monitoring C. Further, let the subplan used by p_c for monitoring s be $p_s(c_s, l_s)$. If p_s is not pareto optimal, then $\exists p'_s(c'_s, l'_s)$ such that ($c_s > c'_s$ and $l_s \geq l'_s$) or ($l_s > l'_s$ and $c_s \geq c'_s$). However, we could then substitute p'_s for p_s in the plan p_c to produce a plan $p'_c(c'_c, l'_c)$ with ($c_c > c'_c$ and $l_c \geq l'_c$) or ($l_c > l'_c$ and $c_c \geq c'_c$) which would contradict the optimality of p_c. Hence, if p_c is optimal, then p_s also has to be pareto optimal. $\square$

The plan generation process continues up to the root of the graph, which then selects the minimum cost plan meeting its latency requirements. This selection at the root also fixes the plans to be used at each node in the graph.

3.3 Plan Execution

Once plan selection is complete, the set of primitive events to monitor are identified and activated. When a primitive event arrives at the base station, it is directed to the corresponding primitive event node. The primitive event node stores the event and then forwards a pointer of the event to its active parents. An active parent is one that has expressed interest in the time interval during which the event arrived. Complex event detection proceeds similarly in the higher level nodes. Each node acts according to its plan upon receiving events either by activating subevents or by detecting a complex event and passing it along to its parents.

A related issue that has been widely discussed in active databases and complex event detection literature [5, 9] is that of *event instance consumption*. An event consumption policy specifies the effects of detecting an event on the instances of that event type's subevents.

Options range from highly-restrictive consumption policies, such as those that allow each event instance to be part of only a single complex event instance, to non-restrictive policies that allow event instances to be shared arbitrarily by any number of complex events. All these consumption policies can be used with our model; although, we assume event instances are shared unless otherwise stated.

4. COST-LATENCY MODELS

Our cost model is not strictly tied to any particular probability model. In addition to the general cost model, we derive cost estimations using two commonly-used parametric probability models: *Poisson* and *Bernoulli* distributions. On the other hand, nonparametric models can be easily plugged-in as well, e.g., histograms can be used to directly calculate the probability values in the general cost model, if the event types do not fit well to common parametric distributions. Model selection techniques, such as Bayesian model comparison [11], can be utilized to select a probability model out of a predefined set of models for each event type. We first assume that event occurrences are independent and then relax this assumption later to capture correlated sequential patterns.

Poisson distributions are widely used for modeling discrete occurrences of events such as receipt of a web request, and arrival of a network packet. A Poisson distribution is characterized by a single parameter λ that expresses the average number of events occurring in a given time interval. In our case, we define λ to be the occurrence rate for an event type in a single time unit. In addition, our initial assumption that events have independent occurrences means that the event occurrences follows a Poisson process with rate λ. When modeling an event type e with the Bernoulli distribution, e has independent occurrences with probability p_e at every time step, provided that the occurrence rate is less than 1.

As described before, an event detection plan consists of a set of states each of which corresponds to the monitoring of a set of events. The cost of a plan is the sum of the costs of its states weighted by state reachability probabilities. The cost of a state depends on the cost of the events monitored in that state. The reachability probability of a state is defined to be the probability of detecting the partial complex event that activates that state. For instance, in Figure 2c, the partial complex event which activates state S_{e_1} is e_1. State reachability probabilities are derived using interarrival distributions of events. When using a Poisson process with parameter λ to model event occurrences, the interarrival time of the event is exponentially distributed with the same parameter. Hence, the probability of waiting time for the first occurrence of an event to be greater than *t* is given by $e^{-\lambda t}$. On the other hand, when using the Bernoulli distribution, the interarrival times have geometric distribution. The reachability probability for initial state is 1 since it is always active and the probability for final state is not required for cost estimation.

For latency estimation, we associate each event type with a latency value that represents the maximum latency its instances can have. Here, we consider identical latencies for all primitive event types for simplicity. However, different latency values can be handled by the system as well. Below, we consider the expected cost and latency of monitoring a simple complex event as an example.

Example: We define the event *and*$(e_1, e_2, e_3; w)$ where e_1, e_2 and e_3 are primitive events with Δt latency and use Poisson processes with rates λ_{e_1}, λ_{e_2} and λ_{e_3} respectively to model their occurrences. First, we consider the naive plan in which all subevents are monitored at all times. Its cost is simply the sum of the occurrence rates of the subevents, $\sum_{i=1}^{3} \lambda_{e_i}$, whereas its latency is the maximum latency among the subevents: Δt. The cost derivation for the three step plan $e_1 \rightarrow e_2 \rightarrow e_3$, given in Figure 2c, is more complex. Using the interarrival distributions to derive reachability probabilities the cost of the three step plan can be derived as:

$$\text{cost for } e_1 \rightarrow e_2 \rightarrow e_3 = \lambda_{e_1} + (1 - e^{-\lambda_{e_1}})2W\lambda_{e_2} + \\ ((1 - e^{-\lambda_{e_1}})(1 - e^{-W\lambda_{e_2}}) + (1 - e^{-\lambda_{e_2}})(1 - e^{-W\lambda_{e_1}}))2W\lambda_{e_3}$$

The plan has an estimated latency of $3\Delta t$ since this is the maximum latency it exhibits (for instance, when the events occur in the order e_3, e_2, e_1 or e_2, e_3, e_1). In the cost equation above and the rest of the paper, we omit the cost terms originating from multiple events occurring in the same time step, assuming without loss of generality that we have a sufficient fine-grained time model.

4.1 Operator-specific models

Below we discuss cost-latency estimation for each operator first for the case where all subevents are primitive and are represented by the same distribution, and then for the more general case with complex subevents. Allowing different probability models for subevents requires using the corresponding model for each subevent in calculating the probability terms, complicating primarily the treatment of the sequence operator, as sums of random variables can no longer be calculated in closed forms.

And Operator. Given the complex event *and*$(e_1, e_2, \ldots, e_n; w)$, a detection plan with $m+1$ states S_1 through S_m, and the final state S_{m+1}, we show the cost derivation using reachability probabilities both for Poisson and Bernoulli distributions below. For event e_j we represent the Poisson process parameter with λ_{e_j} and the Bernoulli parameter with p_{e_j}.

The general cost term for `and` with n operands is given by $\sum_{i=1}^{m} P_{S_i} \times cost_{S_i}$ where P_{S_i} is the state reachability probability for state S_i and $cost_{S_i}$ represents the cost of monitoring subevents of state S_i for a period of length $2W$. In the case that all subevents are primitive $cost_{S_i} = \sum_{e_j \in S_i} 2W\lambda_{e_j}$ when Poisson processes are used and $cost_{S_i} = \sum_{e_j \in S_i} 2Wp_{e_j}$ for Bernoulli distributions.

P_{S_i}, the reachability probability for S_i, is equal to the occurrence probability of the partial complex event that causes the transition to state S_i. For this partial complex event to occur in the "current" time step, all its constituent events need to occur within the last W time units with the last one occurring in the current time step (otherwise the event would have occurred before). Then, P_{S_i} is 1 when *i* is 1 and for $m \geq i > 1$ is given for Poisson processes (i) and Bernoulli distributions (ii) by:

$$\text{(i)} \quad \sum_{e_j \in \bigcup_{k=1}^{i-1} S_k} (1 - e^{-\lambda_{e_j}}) \prod_{\substack{e_t \neq e_j \\ e_t \in \bigcup_{k=1}^{i-1} S_k}} (1 - e^{-\lambda_{e_t} W})$$

$$\text{(ii)} \quad \sum_{e_j \in \bigcup_{k=1}^{i-1} S_k} p_{e_j} \prod_{\substack{e_t \neq e_j \\ e_t \in \bigcup_{k=1}^{i-1} S_k}} (1 - (1 - p_{e_t})^W)$$

Under the identical latency assumption, the latency of a plan for *and* operator is defined by the number of the states in the plan (except the final state). Hence, the latency of a plan for the event *and*$(e_1, e_2, \ldots, e_n)$ can range from Δt to $n\Delta t$.

Sequence Operator. We can consider the same set of plans for `seq` as well. However, sequence has the additional constraint that events have to occur in a specific order and must not overlap. Therefore, the time interval to monitor a subevent depends on the occurrence times of other subevents and is at most W time units.

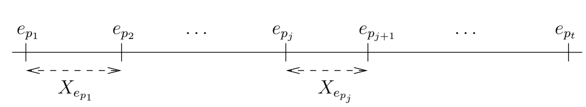


Figure 3: **subevents for** $seq(e_{p_1}, e_{p_2}, \ldots, e_{p_t}; w)$

The expected cost of monitoring the complex event $seq(e_1, e_2, \ldots, e_n; w)$ using a plan with $m+1$ states has the same form $\sum_{i=1}^{m} P_{S_i}$

$\times cost_{S_i}$. Let $seq(e_{p_1}, e_{p_2}, \ldots, e_{p_t}; w)$ with $t \leq n$ and $p_1 < p_2 < \ldots < p_t$ be the partial complex event consisting of the events before state S_i, i.e. $\cup_{k=1}^{i-1} S_k = \{e_{p_1}, e_{p_2}, \ldots, e_{p_t}\}$. Then

1. P_{S_i}, the reachability probability for S_i, is equal to detecting $seq(e_{p_1}, e_{p_2}, \ldots, e_{p_t}; w)$ at a time point. For this complex event to occur subevents has to be detected in sequence as in Figure 3 within W time units. We define the random variable $X_{e_{p_j}}$ to be the time between $e_{p_{j+1}}$ and the occurrence of e_{p_j} before $e_{p_{j+1}}$ (see Figure 3). Then, $X_{e_{p_j}}$ is exponentially distributed with $\lambda_{e_{p_j}}$ if we are using Poisson processes, or has geometric distribution with $p_{e_{p_j}}$ when using Bernoulli distributions.

   For the Poisson process case, we have $P_{S_i} =$ (1-$e^{-\lambda_{e_{p_t}}}$) (1-R(W)) where R(W) = P($\sum_{j=1}^{t-1} X_{e_{p_j}} \geq$ W). A closed form expression for $R(W)$ for the case all exponential variables have distinct parameters is available [13]. For the Bernoulli distribution $P_{S_i} = p_{e_{p_t}}(1 - R(W))$ where $R(W)$ is defined on a sum of geometric random variables. In this case, there is no parametric distribution for $R(W)$ unless the parameters of geometric random variables are identical. Hence, it has to be numerically calculated.

2. Any event e_{i_k} of state S_i should either occur (i) between e_{p_j} and $e_{p_{j+1}}$ for some j or (ii) before e_{p_1} or after e_{p_t} depending on the order in $seq(e_1, e_2, \ldots, e_n; w)$. In case *i*, we need to monitor e_{i_k} between e_{p_j} and $e_{p_{j+1}}$ for $X_{e_{p_j}}$ time units (see Figure 3). For case *ii* we need to monitor the event for $W - \sum_{j=1}^{t-1} X_{e_{p_j}}$ time units. In the cost estimation, we use the expectation values E[$X_{e_{p_j}} \mid \sum_{k=1}^{t-1} X_{e_{p_k}} \leq W$] and $W -$ E[$\sum_{k=1}^{t-1} X_{e_{p_k}} \mid \sum_{k=1}^{t-1} X_{e_{p_k}} \leq W$] for estimating $L_{e_{i_k}}$, the monitoring interval. Then $cost_{S_i}$ is $\sum_{e_{i_k} \in S_i} L_{e_{i_k}} \lambda_{e_{i_k}}$ with Poisson processes and $\sum_{e_{i_k} \in S_i} L_{e_{i_k}} p_{e_{i_k}}$ with Bernoulli distributions.

The latency of a plan for sequence depends only on the latency of the events that are in the same state with the last event (e_n) or are in later states if we ignore the unlikely cases where the latency of the events in earlier states are so high that the last event might occur before they are received at base. If the event $seq(e_1, e_2, \ldots, e_n; w)$ is being monitored with an m-step plan where the j^{th} step contains e_n, then its latency is $(m - j + 1)\Delta t$. This latency difference between `and` and `seq` exists because unlike `seq`, with `and` any of the subevents can be the last event that causes the occurrence. This discontinuity in latency introduced by the last event of sequence operator seems to create an exception for the DP algorithm as the pareto optimal substructure property depends on non-decreasing latency values for the plans formed from smaller subplans. However, in such cases, the pareto optimal plans will include only the minimum cost subplans for monitoring the events in earlier states than e_n, and because one of the minimum cost plan(s) will always be pareto optimal, DP will still find the optimal.

Negation Operator. In our system, negation can be used on the primitive events inside *and* and *seq* operators. Here, we consider the plans for complex events, with negated terms, specified using *and* operator over primitive events. The plans we consider for such events (in addition to the naive plan) resemble a filtering approach. First, we detect the partial complex event consisting of non-negated events only. When that complex event is detected, we monitor the negated events. The detection plan for the complex event defined by non-negated events can be any arbitrary plan discussed for *and* operator. Same set of detection plans can be considered for negated events as well. However, the execution has to be changed in a way that the absence of an event is now what is aimed for. The cost estimations discussed for *and* operator can be applied here by changing the occurrence probabilities with nonoccurrence probabilities.

Finally, to generate plans for events involving the negation operator we made some changes to the plan generation algorithms so that they only produce the plans mentioned above. Basically both plan generation algorithms has been modified such that at any point during their execution the set of plans generated is restricted to the subset of plans fitting the described criteria.

Or Operator. As discussed before, `or` generates a complex event for every event instance it receives. Hence, the only detection plan for *or* operator is the *naive* plan. The cost of the naive plan is the sum of the costs of the subevents and its latency is the highest latency among the subevents.

Generalization to the complex subevents: Given a plan for a complex event, E, we are given a specific plan to use in monitoring each subevent and an order for monitoring them. For the complex subevents E, which generally provide multiple monitoring plans, this means that a particular plan among the available plans is being considered. Also as the occurrence probability of a subevent is independent of the plan it is being monitored with, the only difference between distinct plans is the latency and cost values.

For `seq`, the presented cost model can be used without any changes. As we do not allow the negation operator to be applied to complex events, we do not make any adjustments for that case. For `and`, minor changes are required for dealing with complex subevents. The `and` operator requires only the end points of complex subevents to be in the window interval. Therefore, the complex subevents could have start times before the window interval and, as such, some of their subevents could originate outside the window interval. As a result the monitoring of the subevents of the complex subevents extend beyond the window interval. In such cases, we calculate an estimated monitoring interval based on the window values of the E and its corresponding complex subevent. The `or` operator requires the same modifications.

4.2 Markov models for sequential patterns

The independent and identical distribution (i.i.d.) assumption for the instances of an event type simplifies the cost model and reduces the required computation for the plan costs. However, for certain types of events the i.i.d. assumption will be severely restrictive. A very general subclass of such event types is the event types involving sequential patterns across time. For instance, some event types such as corrupted bits in network transmissions could show *bursty behaviour*.

To illustrate the effects of event types with sequential behaviour on the cost model and the resulting plans we provide the following example scenario, which we verified experimentally. Consider three primitive event types e_1, e_2 and e_3, where e_1 exhibits bursty behaviour, and the complex event $and(e_1, e_2, e_3 \,; w)$. Assume that e_1 has a lower occurrence rate compared to e_2 and e_3 has a very high occurrence rate. When the cost model makes the i.i.d. assumption for the instances of e_1 and the occurrence rates of e_1 and e_2 are high enough, it finds that the partial complex event $and(e_1, e_2;\, w)$ has a relatively high occurrence rate. Therefore, it decides to use the naive plan as no multi-step plan seems to provide lower cost. However, when we use a Markov model (as described below) for modeling the bursty behaviour of e_1, the cost model finds out that the expected occurrence rate of the event $and(e_1, e_2;\, w)$ is much lower than what was found with the i.i.d assumption for e_1. As a result, it chooses the 2-step plan $e_1, e_2 \rightarrow e_3$ that reduces the cost since e_3 has a very high occurrence rate.

We consider using Markov models for modeling such event types with sequential patterns as recent instances of events are likely to be more revealing and not all the previous event instances are relevant. Here, we discuss an m^{th} order discrete-time homogeneous Markov chain in which the occurrence of the event in the current time step depends only on the last m time steps. This model builds on the described Bernoulli cost model. Other probability models could require continuous time Markov models to be used.

Denoting the occurrence of the event type e_1 at time t as a binary random variable e_1^t that has the value 1 when the event occurs, we have $P(e_1^t|e_1^1, e_1^2, .., e_1^{t-1}) = P(e_1^t|e_1^{t-m}, .., e_1^{t-1})$. Such a Markov chain of order m can be represented as a first order Markov chain by defining a new variable y as the last m values of e_1 so that the chain follows the well-known Markov property. Then, we can define the Markov chain by its transition matrix, P, mapping every possible values of the last m time steps to the possible next states. The stationary distribution of the Markov chain, $\bar{\pi}$, can then be found by solving $\bar{\pi}P = \bar{\pi}$. In this case, modifying the cost model to use the Markov chain requires one to simply use the transition matrix for calculating the state reachability probabilities and $\bar{\pi}$ as the occurrence probability of the event at a time step. We omit the details of the modifications here due to space constraints.

5. OPTIMIZATION EXTENSIONS

5.1 Optimizing for Shared Subevents

The hierarchical nature of complex event specification may introduce common subevents across complex events. For example, in a network monitoring application we could have the *syn* event indicating the arrival of a TCP *syn* packet. Various complex events could then be specified using the *syn* event, such as syn-flood (sending syn packets without matching acks to create half-open connections for overwhelming the receiver), a successfull TCP session, and another event detecting port scans where the attacker looks for open ports.

The overall goal of plan generation is to find the set of plans for which the total cost of monitoring all the complex events in the system is minimized. The plan generation algorithms presented in Section 3.2 do not take the common subevents into account as they are executed independently for each event operator in a bottom-up manner. As such, while the resulting plans minimize the monitoring cost of each complex event separately, they do not necessarily minimize the total monitoring cost when shared events exist. Here, we modify our algorithm to account for the reduction in cost due to the common subevents and to consider the plans that facilitate sharing to further reduce cost when possible.

To estimate the cost reduction due to sharing, we need to find out the expected amount of sharing on a common subevent. However, the degree of sharing depends on the plans selected by the parents of the shared node, as the monitoring of the shared event is regulated by those plans. Since the hierarchical plan generation algorithm (Section 3.2.3) proceeds in a bottom-up fashion, we cannot identify the amount of sharing unless the algorithm completes and the plans for all nodes are selected. To address these issues, we modify the plan generation algorithm such that it starts with the independently selected plans and then iteratively generates new plans with increased sharing and reduced cost. The modified algorithm is given in Algorithm 2 for the case of a single shared event.

Algorithm 2 Plan generation with a shared event

1. s = shared event, A = s.parents
2. P = $0^{|A|}$ // zero vector of length $|A|$
3. plans = generatePlans() // execute hierarchical plan generation
4. // from Section 3.2.3
5. **for all** a $\in$ A **do**
6. q = the plan involving a in plans
7. P[a]= cost of s in q / occurrence rate of s
8. **for all** q $\in$ plans **do**
9. q.cost = q.cost $-$ (cost of s in q)
10. $+$ (shared cost of s under P with q)
11. isLocalMinimum = **false**, P$' = 0^{|A|}$
12. **while** !isLocalMinimum **do**
13. newplans = generatePlans(A,P)
14. **for all** a $\in$ A **do**
15. q = the plan involving a in newplans
16. P$'[a]$ = cost of s in q / occurrence rate of s
17. **for all** q $\in$ newplans **do**
18. q.cost = q.cost $-$ (cost of s in q)
19. $+$ (shared cost of s under P$'$ with q)
20. **if** newplans.cost $>$ plans.cost || newplans $==$ plans **then**
21. isLocalMinimum = **true**
22. **else**
23. plans = newplans, P = P$'$

After the independent plan generation is complete (line 3), each node will have selected its plan, but the computed plan costs will be incorrect as sharing has not been considered. To fix the plan costs, first for each parent of shared node we calculate the probability that it monitors the shared event in a given time unit (lines 5-7). We have already computed this information during the initial plan generation as the plan costs involve the terms: *probability of monitoring the shared node* $\times$ *occurrence rate of the shared event*. We can obtain these values with little additional bookkeeping during plan generation. Next, using the probability values, we adjust the cost of each plan to only include the estimated shared cost for the common subevent (lines 8-10). We assume the parents of the shared node function independently and fix the cost for the cases where the shared event is monitored by multiple parents simultaneously.

Then we proceed to the plan generation loop during which at iteration new plans are generated for the nodes starting from the parents of the shared node. However, in this execution of the plan generation algorithm (line 13), for each operator node, the algorithm computes the reduction in plan costs due to sharing by using the previous shared node monitoring probabilities, P, and updating the shared node monitoring probability with each plan it considers. Hence, the ancestors of the shared node may now change their plans to reduce cost. Moreover, the new plans generated in each iteration are guaranteed to increase the amount of sharing if they have lower cost than the previous plans. This is because the plan costs can only be reduced by monitoring the shared node in earlier states. The algorithm iterates till a plan set with a local minimum total cost is reached, although there is no guarantee to achieve the global minimum. We consider it future work to study techniques such as simulated anealing and tabu search [12] for convergence to global minimum cost plans for the case with shared events.

5.2 On Further Constraint Optimization

We now briefly describe how spatial and attribute-based constraints affect the occurrence probabilities of events and discuss additional optimizations in the presence of these constraints. A comprehensive evaluation of these techniques is outside the scope of this paper.

First, we consider **spatial constraints** that we define in terms of regional units. The space is divided into regions such that events in a given region are assumed to occur independently from the events in other regions. The division of space into such independent regions is typical for some applications. For instance, in a security application we could consider the rooms (or floors) of a building as independent regions. In addition, it is also easy for users to specify spatial constraints (by combining smaller regions) once regional units are provided. An alternative would be to treat the spatial domain as a continuous ordered domain of real-world (or virtual) coordinates and then perform region-coordinate mappings. This latter approach would allow us to use math expressions and perform optimizations using spatial-windowing constraints, similar to what we described for temporal constraints.

The effects of spatial constraints on event occurrence probabilities can be incorporated in our framework with minor changes. First, we modify our model to maintain event occurrence statistics per each independent region and event type. Then, when a spatial constraint on a complex event is given, we only need to combine the information from the corresponding regions to derive the associated event occurrence probability. For example, if we have Poisson processes with parameters λ_1 and λ_2 for two regions, then the Poisson process associated with the combined region has the parameter $\lambda_1 + \lambda_2$. Hence, by combining the Poisson processes we can easily construct the Poisson process for any arbitrary combination of independent regions. If the regions are not independent, we need to derive the corresponding joint distributions. A interesting optimization would be to use different plans for monitoring different spatial regions if doing so reduces the overall cost.

Attribute-based constraints can be optimized using standard histogram based techniques. Histograms provide the information for deriving the selectivity probabilities of attribute-based constraints, which we can then use to derive the event occurrence probabilities. Moreover, value based attribute constraints can be pushed down to event sources, further reducing the number of transmitted events. Parameterized attribute constraints between events can also be pushed down whenever one of the events is monitored earlier than the other one.

6. EXPERIMENTAL EVALUATION

In this section, we characterize the performance of our algorithms and investigate the effects of various factors through a set of experiments. We implemented all our algorithms in Java. Specific hardware configurations we used for experimentation is not relevant as we report abstract metrics that do not depend on the run-time environment.

Our experiments involve both synthetic and real-world data sets. For synthetic data sets, we used the *Zipfian* distribution (with default skew = 0.255) to generate event occurrence frequency parameter values, which are then plugged into the exponential distribution to generate event arrival times. The real data set we used is a collection of network traffic logs obtained from the Planetflow web site [18].

The primary performance metric we show is the ”transmission factor”, which represents the ratio of the number of primitive events received at the base to the total number of primitive events generated by the sources. This metric quantifies the extent of event suppression our plan-based techniques can achieve over the standard push-based approach. We also present the ”minimum transmission factor”, the ratio of the number of primitive events that participate in the complex events that actually occurred to the total number generated. This metric represents the *theoretical best* that can be achieved and thus serves as a tight lower bound on transmission costs. All the experiments involving synthetic data sets are repeated till results statistically converge with approximately 1.2% average and 5% maximum variance. Error bars are shown when the variance exceedeed this threshold.

6.1 Single-Operator Analysis

We first analyze in-depth the base case where our complex events consist of individual operators.

On window size and detection latency: We defined the complex events `and(`e_1, e_2, e_3`)` and `seq(`e_1, e_2, e_3`)`, where e_1, e_2 and e_3 are primitive events. We ran both the dynamic programming (DP) and heuristic-based algorithms for different windows sizes and plan lengths (as an indication of execution plan latency). The results are shown in Figures 4(a) and 4(b).

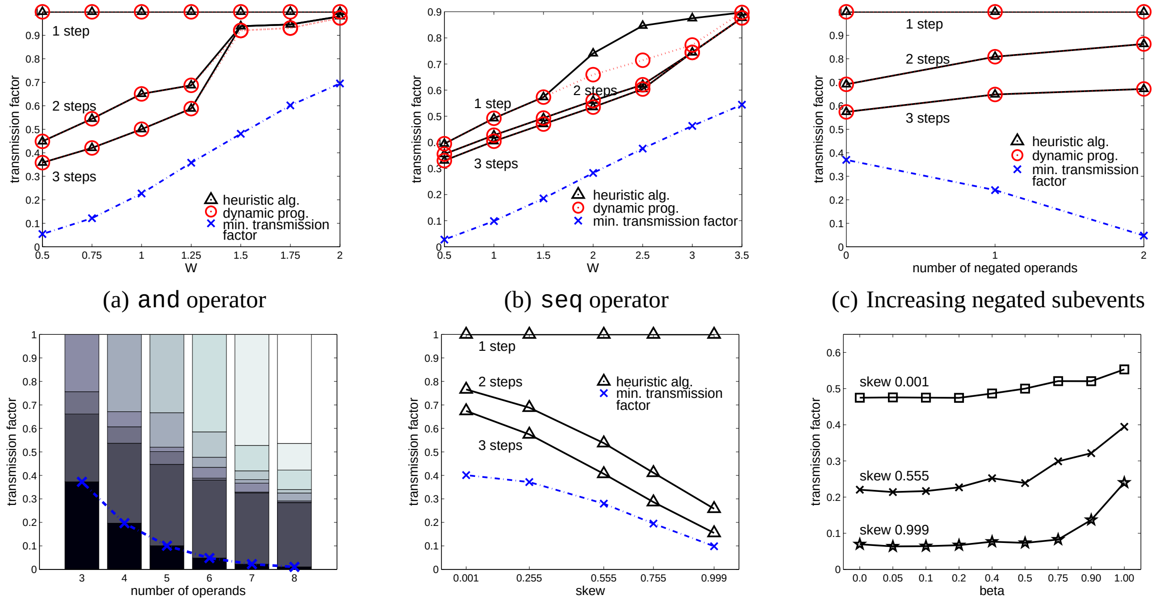
Results reveal that, as the number of steps in the plan increases (from 1 to 3 in this case), the event detection cost generally decreases. In the case of the `and` operator, the heuristic method finds the optimal solution (as does the dynamic programming algorithm), as we are considering a trivial complex event. However, in the case of the `seq` operator, there is some difference between the two algorithms for the 1-step case. Recall that due to the ordering constraint, the `seq` operator does not need to monitor the later events of the sequence unless the earlier events occur. Therefore, it can reduce the cost even under hard latency requirements. However, this asymmetry introduced by the `seq` operator is also the reason why our heuristic algorithm fails to produce the optimal solution. Finally, the event detection costs tend to increase with increasing window sizes since larger windows increase the probability of event occurence. If the window is sufficiently large, the system would expect the complex event to occur roughly for each instance of a primitive event type. Consequently, the system will choose to monitor all the events simultaneously and relaxing the latency target does not help reduce the cost.

Effects of Negation: We performed an experiment using a single operator `and(`e_1, e_2, e_3`)` in which we varied the number of negated subevents. Results are given in Figure 4(c). We observe that the cost increases with more negated subevents, although fewer complex events are detected. This is mainly because (1) all the transmitted non-negated subevents have to be discarded when a negated subevent that prevents them from forming a complex event is detected, and (2) as described in Section 4, the monitoring of the negated and non-negated events is not interleaved: the negated subevents are monitored only after the non-negated subevents. Results are similar for uniformly distributed event frequencies (yet the cost seems to be more independent of the number of negated subevents in the uniform case). For highly-skewed event frequencies, the results depend on the particular frequency distribution. For instance, if the frequency of the negated event (or one of the negated events) is very high, then the complex event almost never occurs, but the monitoring cost is also low since other events have low frequencies.

Increasing the operator fanout: We now analyze the relation between the cost and the fanout (i.e., number of subevents) using an `and` operator. To eliminate the effects of frequency skew, we used uniform distribution to produce the event frequencies. Results from running the heuristic algorithm (DP results are similar) are shown in Figure 4(d), in which the lowest dark portion of each bar shows the minimal transmission factor and the cost values for increasingly strict deadlines are stacked on top each other. We can see that (i) increasing the fanout tends to decrease the number of detected complex events and (ii) larger fanout implies we have a wider latency spectrum, thus a larger plan space and more flexibility to reduce cost.

Effects of frequency skew: In this experiment, we define the complex event `and(`e_1, e_2, e_3`)` with a temporal window of size of 1 and vary the parameter of the Zipfian distribution with which event frequencies are generated. The total number of primitive events for different event frequency values are kept constant. Figure 4(e) shows that a higher number of complex events is detected with low-skew streams and the cost is thus higher. Furthermore, our algorithms can effectively capitalize on high-skew cases where there is significant difference between event occurence frequencies by postponing the monitoring of high-frequency events as much as the latency constraints allow.

Tolerance to statistical estimation errors: We now analyze the effects of parameter estimation accuracy on system performance using $and(e_1, e_2, \ldots, e_5)$, where $e_1, e_2, \ldots, e_5$ are primitive events,



(a) `and` operator (b) `seq` operator (c) Increasing negated subevents

(d) Increasing operands (e) Increasing skew (f) Tolerance to estimation errors

Figure 4: Operator wise experiments

and a window of size 1. We use the Zipfian distribution to create the "true" occurrence rates $\lambda^T = [\lambda^T_{e_1}, \lambda^T_{e_2}, \ldots, \lambda^T_{e_5}]$ for the events. We then define λ^β with $\lambda^\beta_i = \lambda^T_i \pm \beta\lambda^T_i$ for $1 \leq i \leq 5$ to be an estimation with error β (the $\pm$ indicates that the error is either added or subtracted based on a random decision for each event). The results are given in figure 4(f).

For highly skewed occurrence rates, the estimation error has a larger impact on the cost as the occurrence rates are far apart in such cases. For very low skew values, error does not affect the cost much since most of the events are "exchangeable", i.e., selected plans are independent of the monitoring order of the events as switching an event with another does not change the cost much. We did a similar experiment using events with many operators instead of a single one. The relative results and averages were similar, however, the variance was higher (approximately 10%), meaning for some complex event instances the cost could be highly affected by the estimation error.

6.2 Effects of Event Complexity

Increasing event complexity: In this experiment, we varied the number of operators used in complex events from 1 to 7. Each operator was given 2 or 3 subevents with equal probability and a window of size 2.5. In figure 5(a), we provide the average event detection costs for the complex events that have approximately the same number of occurrences (as shown by the minimum transmission factor curve). We can see that the cost does not depend on the number of operators in the expression and but instead depends on the occurrence frequency of the complex event. On the other hand, as the number of operators in an expression is increased, the occurrence probability of the event decreases.

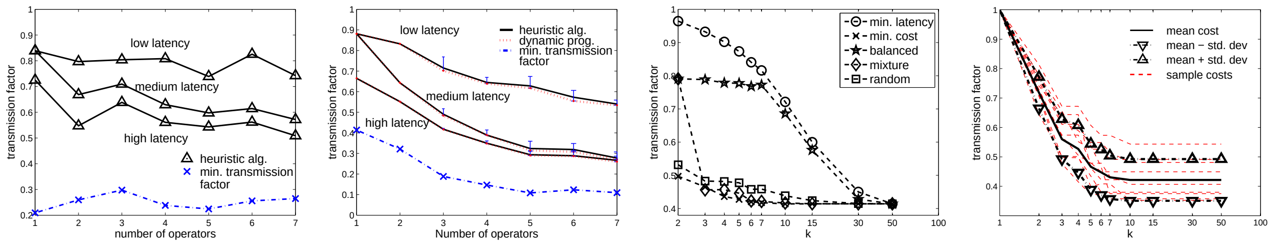
Dynamic programming vs. heuristic plan generation: The average event detection costs for both plan generation algorithms are given in figure 5(b). The results, obtained using the same parameter settings from the previous example, show that the heuristic method performs, on average, very close to the dynamic programming method. The error bars indicate the standard deviation of the difference between the two cost values.

Selective hierarchical plan propagation: In this experiment, we analyze the effects of the parameter k, which limits the number of plans propagated by operator nodes to their parents during hierarchical plan generation (see section 3.2.1). We defined complex events using exclusively `and` operators, each with a fixed window size of 2.5, and together forming a complete binary tree of height 4. We consider the following strategies for picking k plans from the set of all plans produced by an operator:

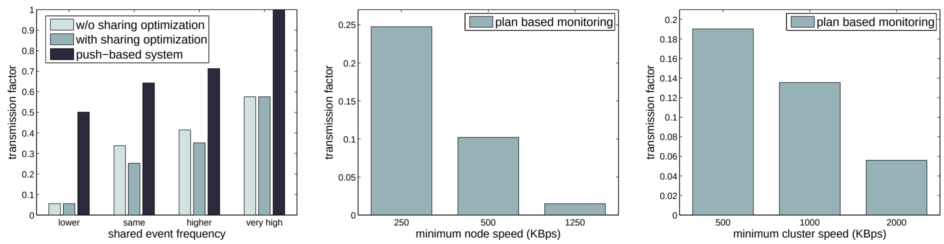
- **random selection**: randomly select k plans from all plans.
- **minimum latency**: pick the k plans with minimum latency.
- **minimum cost**: pick the k plans with minimum cost.
- **balance cost and latency**: represent each plan in the $\Re^2$ (cost, latency) space, then pick the k plans with minimum length projections to the $cost = latency$ line.
- **mixture**: pick k/3 plans using the minimum latency strategy, k/3 using the minimum cost strategy and the other k/3 plans using the balanced strategy.

The average cost of event detection for each strategy with different k values are given in figure 5(c) in which DP is used. Greater values of k generally means reduced cost for event detection since increasing the value of k helps us get closer to the optimal solution. The mixture and the minimum cost strategies perform similarly and approach the optimal plan even for low values of k. However, the minimum cost strategy does not guarantee finding a feasible plan for each complex event since it does not take the plan latency into account during plan generation. On the other hand, the mixture strategy will find the feasible plans if they exist since it always considers the minimum latency plans.

We repeated the same experiment with the heuristic plan generation method using the mixture strategy (figure 5(d)). Results are similar to the DP case; however, the heuristic algorithm, unlike the DP algorithm, does not produce the set of all pareto optimal plans. Moreover, the size of the plan space explored by the heuristic algorithm depends on the number of moves it can make without reaching a point where no more moves are available. Therefore, even when the value of k is unlimited, the heuristic method does not guarantee optimal solutions, which is not the case with the DP approach.



(a) Increasing the #operators (b) DP vs. heuristic planning (c) Plan selection methods (d) Selective plan propogation



(e) Leveraging sharing (f) Load spike event (g) Suspicious activity event

Figure 5: Event complexity, shared optimization, plan generation and PlanetLab experiments

6.3 Effects of Event Sharing

To quantify the potential benefits of leveraging shared subevents across multiple complex events, we generated two complex events with a common subevent tree and compared the performance with and without shared optimization. Each complex event has 3 `and` operators, one of which is shared. The shared operator has 6 primitive events, 2 of which are common to both complex events. In the experiment, we varied the frequency of the complex event that corresponds to the shared subtree. In Figure 5(e), we see that when the frequency of the shared part is low, leveraging sharing does not lead to a noteworthy improvement since the shared part is chosen to be monitored earlier in both cases anyway. When the frequency of the shared part is the same with or slightly higher than the non-shared parts, the latter are monitored earlier without sharing optimization. In this case, shared optimization reduces the cost by monitoring the shared part first. Finally, when the shared part has very high frequency, non-shared parts are monitored first in both cases. Shared optimization allows for a better estimation of the shared subevent costs, which can cause better execution plans to be generated when using heuristics for plan generation. If we used exhaustive plan generation, which is not tractable in general, we choose identical plans both with and without sharing.

6.4 Experiments with the PlanetLab Data Set

The PlanetLab data set we used consists of 5 hours of network logs (1pm-6pm on 6/10/2007) for 49 PlanetLab nodes [18]. The logs provide aggregated information on network connections between PlanetLab nodes and other nodes on the Internet. For each connection, indicated by source and destination IP/port pairs, the information includes the start and end times, the amount of generated traffic and the network protocol used. We experimented with a variety of complex events commonly used in network monitoring applications. Here, we present the results for two representative complex events that can be applied over the PlanetLab data.

Capturing load spikes: We define a PlanetLab node as (i) *idle* if its average network bandwidth consumption (incoming and outgoing) within the last minute is less than 125KBps and as (ii) *active* if the average speed is greater than a threshold T. The *spike* event monitors for the following overall network load change: the event that more than half of all nodes are idle, followed by the event that more than half is active within a specified time interval. Thus, the complex event is defined as `seq`*(count(idle)* $>$ *%50 of all nodes, count(active)* $>$ *%50 of all nodes; w=30min)*. Note here that the *count* operator is evaluated in an entirely push-based manner and thus does not affect plan generation or execution. The results are provided in Figure 5(f) for T = 250, 500, and 1250 KBps. We see substantial savings that range from 75% to 97%, respectively.

Active-diverse clusters: For this experiments, we express a complex event inspired by Snort [20]. The basic idea is to identify a cluster of machines that exhibit high traffic activity (active) *and* connect over a large number of protocols (diverse) within a time window.

The complex event is graphically illustrated in Figure 6. We define a cluster to be a set of machines from the same $/8$ IP class. A *diverse cluster* is defined as a cluster with more than C=500 connections to PlanetLab nodes in total within the last minute (multiple connections from the same IP address are counted distinctly). To specify this complex event we first define a *locally diverse cluster* event which monitors the event that a PlanetLab node has more than $\frac{C}{N=49}$ connections with a cluster. The diverse cluster complex event is specified as *sum(conns)*$>$ C *group by cluster*. Then, it is `and`'ed with the locally diverse cluster event which acts as a prerequisite for the diverse cluster event and helps reduce monitoring cost. Next, using the diverse cluster event, we define the *unexpected diverse cluster* event as the diverse cluster event preceded by no occurrences of the event that the same cluster has more than C/2 connections within the last 5 minutes. Moreover, we define the active cluster event, similar to the diverse cluster event, but thresholding on the network traffic ($\sum$ traffic $> T$) instead of the connections. Finally, we define the top level complex event as the `and` of the active cluster and unexpected diverse cluster events to monitor for the active clusters that has recently made many connections.

Figure 5(g) shows the results for three threshold values. In all cases, we observe significant savings that increase with increasing thresholds. The primary reason for this behavior is that the complex event and its subevents become less likely to happen as we increase the threshold, thereby yielding increasingly more savings for our plan-based approach that stages the event monitoring process.

7. RELATED WORK

In continuous query processing systems such as TinyDB [2] for wireless sensor networks, and Borealis [15] for stream processing applications queries are expected to constantly produce results. Push

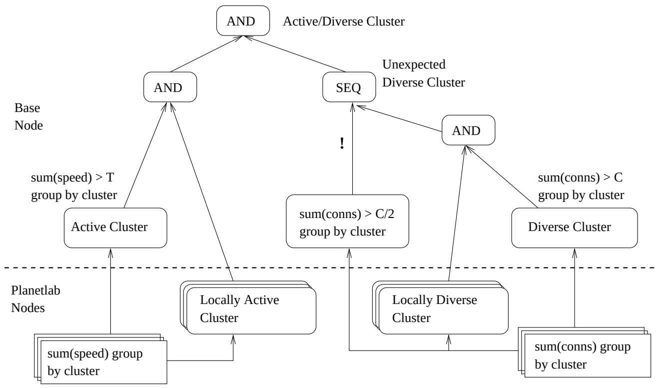


Figure 6: Active/Diverse cluster event specification

based data transfer, either to a fixed node or to an arbitrary location in a decentralized structure, is characteristic of such continuous query processing systems. On the other hand, event detection systems are expected to be silent as long as no events of interest occur. The aim in event systems is not continuous monitoring of the data, but is the detection of events of interest.

In the active database community, ECA (event-condition-action) rules have been studied for building triggers [8]. Triggers offer the event detection functionality through which database applications can subscribe to in-database events, e.g. the insertion of a tuple. However, most in-database events are simple whereas more complex events could be defined in the environments we consider. Many active database systems such as Samos [4], Ode Active Database [5], and Sentinel [6] have been produced as the results of the studies in the active database area. Most systems provide their own event languages. These languages form the base of the event operators in our system. However, our event operators have an additional windowing construct.

In the join ordering problem, database query optimizers try to find ordering of relations for which intermediate result sizes is minimized [19]. Most query optimizers only consider the orders corresponding to left-deep binary trees mainly for two reasons: (1) Available join algorithms such as nested-loop joins tend to work well with left-deep trees, and (2) Number of possible left-deep trees is large but not as large as number of all trees. Our problem of constructing minimum cost monitoring plans is different from the join ordering problem for the following reasons. First, we are not limited to binary trees since multiple event types can be monitored in parallel. Second, our cost metric is the expected number of events sent to base. Finally, we have an additional constraint, i.e. the latency constraint, further limiting the solution space.

In high performance complex event processing [7] optimization methods for efficient event processing are described. There the aim is to reduce processing cost at the base station. While our system also helps reduce the processing cost, our main goal is to minimize the network traffic. Moreover, the proposed system does not consider distributed event processing, and simultaneous queries.

Event processing has also been considered in event middleware systems which are natural extensions to the publish/subscribe systems. In Hermes [3], a complex event detection module has been implemented and an event language based on regular expressions is described. Decentralized event detection is also discussed. However, plan based event detection is not considered. In [14], authors describe model based approximate querying techniques for sensor networks. Similar to our work, plan based approaches to data collection has been considered for network efficiency. Authors also discuss confidence based results and consider dependencies between sensor readings in their data model.

8. CONCLUSIONS AND FUTURE WORK

CED is a critical capability for emerging monitoring applications. While earlier work primarily focused on optimizing processing requirements, our effort is towards optimizing communication needs using a plan-based approach when distributed sources are involved. To our knowledge, we are the first to explore cost-based planning for CED.

Our results, based on both artificial and real-world data, show that communication requirements can be substantially reduced by using plans that exploit temporal constraints among events and statistical event models. Specifically, the big benefits came from a novel multi-step planning technique that enabled "just-enough" monitoring of events. We believe some of the techniques we introduced can be applied to CED on even centralized disk-based systems (i.e., to avoid pulling all primitive events from the disk)

This is a rich research area with many open problems. Our immediate work will explore probabilistic plans for sensor-based applications and augmenting manual event specifications with learning.

9. REFERENCES

[1] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, Albert Vernon. Scalable Trigger Processing. ICDE 1999: 266-275.
[2] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinydbTransactions on Database Systems (TODS), 2005.
[3] Peter R. Pietzuch. "Hermes: A Scalable Event-Based Middleware". Ph.D. Thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.
[4] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In Proc. 4. Intl. Workshop on Research Issues in Data Engineering, 1994.
[5] S. Chakravarthy. et al. Composite Events for Active Databases: Semantics, Contexts and Detection, VLDB 1994.
[6] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Data and Knowledge Engineering, 14(10):1–26, 1994.
[7] Eugene Wu, et al. High-Performance Complex Event Processing over Streams. SIGMOD 2006
[8] N. Paton and O. Diaz, 'Active Database Systems', ACM Comp. Surveys, Vol. 31, No. 1, 1999.
[9] Zimmer, D. and Unland, R. On the Semantics of Complex Events in Active Database Management Systems. ICDE'99.
[10] The Power of Events. David Luckham, May 2002.
[11] Pattern Recognition and Machine Learning. Bishop, Christopher M. 2006, ISBN: 978-0-387-31073-2.
[12] Combinatorial optimization: algorithms and complexity. Christos H. Papadimitriou, Kenneth Steiglitz. 1998.
[13] S. V. Amaria and R. B. Misra, Closed-form expressions for distribution of sum of exponential random variables, IEEE Trans. Reliability, vol. 46, no. 4, pp. 519-522, Dec. 1997.
[14] Amol Deshpande, et al. Model-based approximate querying in sensor networks. VLDB J. 14(4): 417-443 (2005)
[15] Daniel Abadi, et al. The Design of the Borealis Stream Processing Engine. CIDR'05.
[16] S. Chandrasekaran, et al. TelegraphCQ: Continuous Dataflow Processing. In ACM SIGMOD Conference, June 2003.
[17] R. Motwani, et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In CIDR Conference, January 2003.
[18] http://planetflow.planet-lab.org/
[19] Selinger, P. G., et al. 1979. Access path selection in a relational database management system. SIGMOD '79.
[20] SNORT Network Intrusion Detection System. http://www.snort.org
[21] S. Li, S. H. Son, and J. A. Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In IPSN 2003