Tech. Report

Department of Computer Science,
National Tsing Hua University

CS-1207-31

Cho, C.W., Y. Zheng, and A.L.P. Chen

CBS-Tree: Event Prediction Using Episode Rules over Event Streams

Abstract. Event prediction over event streams is an important problem with broad applications. For this problem, rules with *predicate events* and *consequent events* are given, and then current events are matched with the predicate events to predict future events. Over the event stream, some matches of predicate events may trigger duplicate predictions, and an effective scheme is proposed to avoid such redundancies. Based on the scheme, we propose a novel approach CBS-Tree to efficiently match the predicate events over event streams. The CBS-Tree approach maintains the recently arrived events as a tree structure, and an efficient algorithm is proposed for the matching of predicate events on the tree structure, which avoids exhaustive scans of the arrived events. By running a series of experiments, we show that our approach is more efficient than the previous work for most cases.

Keywords: Continuous query, episode rules, minimal occurrence, event stream, prediction.

1 Introduction

In many applications, *events* such as stock fluctuations in the stock market [12], alarms in telecommunication networks [10], and road conditions in traffic control [9] often arrive rapidly as a continuous stream. Particular events on this kind of streams are important to applications but may expire in a short period of time. Therefore, an effective mechanism that predicts the incoming events based on the past events is helpful for the quick responses to particular events. An example is the forecast of traffic jams according to the nearby road conditions. In daily rush hours, the avoidance of all traffic jams is essential to traffic control. In general, there are two main steps to enable effective prediction over event streams. The first step is to obtain the causal relationships among events, which can be either derived from the past events or given by users. The second step is to keep monitoring the incoming events and utilize these relationships for online prediction. In this paper, we consider the causal relationships in the form of *rules* and aim at designing an efficient mechanism for the second step. In the following, we first illustrate the form of rules in our consideration by a traffic control example.

Figure 1 shows a map with four checkpoints denoted as a, b, c, and d. Suppose each checkpoint is associated with a sensor that can periodically report its traffic condition, including the *flow* and *occupancy* [9]. The *flow* is the number of cars passing by a sensor per minute. The *occupancy* is a ratio of one-minute interval in which the detection zone is occupied by a car. If the sensor of a road reports low flow and high occupancy, the traffic of this road can be regarded as "congested." An example rule, represented in the form of $\alpha \Rightarrow \beta$, is shown in Figure 2, where α is called the *predicate* and β the *consequent*. The predicate stands for a set of events that constitute a directed acyclic graph. Each vertex in the graph represents an event, while each edge from vertex u to vertex v indicates that the event u must occur before the event v. For instance, the predicate in Figure 2 makes it a condition that the two events sensing high flow and low occupancy on segments a and b respectively precede the event sensing high flow and high occupancy on segment c[1]. Note that this predicate allows the events on a and b to occur in any order. The consequent is a single event and the rule means that the

[1]We refer to the high flow and low occupancy as sufficient conditions for "smooth driving," while the high flow and high occupancy mean "smooth driving with a large number of passing cars."

consequent will appear in the near future after the predicate shows up.

Furthermore, two time bounds are assigned to the rule and its predicate, respectively. The time bound on the predicate, called *predicate window*, is a temporal constraint on the occurrences of all the events in the predicate, while the one on the entire rule, called *rule window*, corresponds to the temporal constraint on the occurrences of all the events in the entire rule. Consider the episode rule in Figure 2 as an example. If all the events in the predicate occur within 10 minutes in accordance with the temporal order specified in the predicate, the event in the consequent will appear with certain probability in no more than 15 minutes. Specifically, segment d will become low flow and high occupancy (i.e., traffic jam occurs in d). Therefore, once we find a mach for the predicate over the traffic event stream, the traffic control system can make a proper response accordingly, e.g., limiting the traffic flow from R_1 to d or guiding the traffic flow on c to R_2, before the traffic jam on d occurs. We call this kind of rules the *episode rule* [11]. The problem we address in this paper is thus formulated below:

Given a set of episode rules, keep monitoring each of them over the stream of events to detect whether all the events in the predicate appear in the order as specified in the predicate of the rule and satisfy the time bound, and return as soon as a match is found the event in the consequent together with the time interval it will appear.

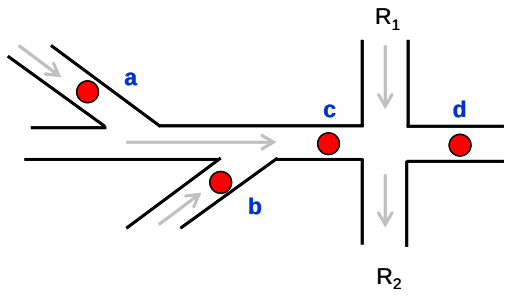


Figure 1: A roadmap

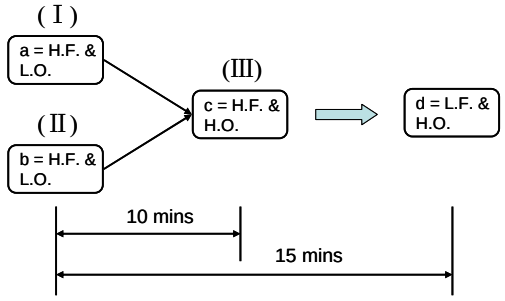


Figure 2: An episode rule

In this paper, we assume that the system receives only one event at a time. Moreover, we consider an event as a set of values from separate attributes and do not explicitly distinguish them by the attributes. Continue the traffic control example in Figure 2. Let W, X, Y, and Z denote the traffic events "a = H.F. & L.O.", "b = H.F. & L.O.", "c = H.F. & H.Q.", and "d = L.F. & H.Q.", respectively. An example stream of traffic events from timestamp 8:00 to 8:07 are depicted in Figure 3, where the events W, X, and Y within the time interval [8:02, 8:05] match the predicate. Thus, it would be reported that event Z will arrive within the time interval (8:02, 8:17). We call the occurrences of the events corresponding to the matches of the predicate as *predicate occurrences*. Let the pair (e, t) denote the event e that occurs at time t. In this example, we say that <(W, 8:02), (X, 8:03), (Y, 8:05)> is a predicate occurrence of the episode rule in Figure 2.

For an episode rule, there can be more than one predicate occurrence in a time period. For example, given the episode rule in Figure 2, in Figure 3 there are three predicate occurrences, O_1=<(X, 8:00), (W, 8:02), (Y, 8:05)>, O_2=<(W, 8:02), (X, 8:03), (Y, 8:05)>, and O_3=<(W, 8:02), (X, 8:03), (Y, 8:07)>. Note that only the time interval of O_2 ([8:02, 8:05]) does not enclose the time interval of any other. We call this kind of occurrences the *minimal occurrence* [11]. The events of a minimal occurrence span a shorter period. From O_2 and the rule window, it can be predicted that event Z will occur within the time interval (8:05, 8:17). We call this time

interval the *predicted interval* of O_2. It can be seen that the predicted intervals of O_1 and O_3 are (8:05, 8:15) and (8:07, 8:17), respectively. Since both the two predicted intervals are enclosed in the predicted interval of O_2, O_1 and O_3 provide less but redundant information for prediction than O_2. Therefore, one of our goals in this paper is to discover all the minimal occurrences for the predicate of every episode rule but ignore the other predicate occurrences. Notice that, many real time applications such as intrusion detection systems concern when the consequent will likely occur, rather than when it will most likely occur. Our work aims at promptly reporting the time interval in which the consequent will occur for these applications.

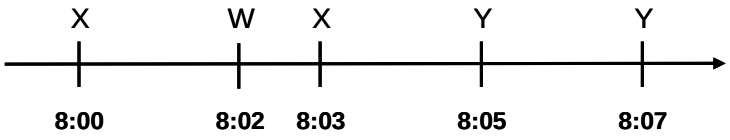


Figure 3: An example stream of traffic events

The burst and endlessness characteristic of data streams make it impossible to retrieve every occurrence and then verify whether it should be reported. Thus, we need an efficient way to ensure that no redundant information is reported (i.e., only minimal occurrences are reported). Cho et al. [4] propose a method named *ToFel* (Topological Forward Retrieval) to match the minimal occurrences from the stream. ToFel takes advantage of the topological characteristic in the predicate to find the minimal occurrences by incrementally maintaining parts of the predicate occurrences, and thus avoids scanning the stream backward. It constructs one event filter with a queue for each predicate to be matched. The filter continuously monitors the newly arrived events and only keeps those that are likely to be parts of the minimal occurrences. However, ToFel may suffer from the plenty of *false alarms* over the stream, i.e., partial matches that do not finally lead to a minimal occurrence. Moreover, since ToFel needs to keep the arrived events for the occurrences of a predicate, the memory required by ToFel is proportional to the given episode rules.

In this paper, we propose a novel approach called *CBS-Tree*. CBS-Tree (Circular Binary Search Tree) maintains the recently arrived events in a tree structure, and an efficient algorithm is proposed for the retrieval of the predicate occurrences upon the tree structure, which avoids scanning the stream repeatedly. The memory requirement in CBS-Tree is only affected by the maximum of all the predicate windows and thus is independent of the amount of episode rules. Moreover, since the occurrence retrieval on CBS-Tree is triggered by the sinks of predicates (the vertices with no edge out of them), the proposed approach does not need to maintain the partial match for every predicate occurrence. Therefore, CBS-Tree outperforms ToFel in processing time when the stream has plenty of false alarms.

The remainder of the paper is organized as follows. Section 2 presents the problem statements including the preliminary and our problem. Section 3 introduces the CBS-Tree method. The experimental results are discussed in Section 4. Section 5 states the related works. Finally, we give a conclusion and the future directions in Section 6.

2 Problem Formulation

In this section, we first define several terms related to our problem, and then the basic concepts used in our approaches are presented.

2.1 Terminology

Episode. An *episode* is a directed acyclic graph *g*, where each vertex stands for an event, and all vertices correspond to distinct events. The event corresponding to vertex v is denoted as $\varepsilon(v)$. Each directed edge (u, v) in *g* indicates that $\varepsilon(u)$ must precede $\varepsilon(v)$. We call this precedence the *temporal order* between vertex u and vertex v and denote it as $u \prec v$. Let $V(g)$ and $E(g)$ respectively denote the vertex set and the edge set of *g*. A vertex *v* is *reachable* from another vertex *u* if there is a directed path that starts from *u* and ends at *v*. The vertex u is a *predecessor* of vertex v if v is reachable from u, and v is a *successor* of u. Moreover, if $(u, v) \in E(g)$, u is a *direct predecessor* of v, and v is a *direct successor* of u. A vertex without any predecessor is the *source* of the episode *g*, while a vertex without any successor is the *sink* of *g*. In other words, there is no edge into the source and no edge out of the sink. There can be more than one sources and sinks in an episode.

Episode occurrence. An *event stream* can be represented as $\hat{S} = <(e_1, t_1), (e_2, t_2), \ldots (e_n, t_n) \ldots>$, where $t_1<t_2< \ldots<t_n \ldots$. Given an *event sequence* $S' = <(e'_1, t'_1), (e'_2, t'_2) \ldots, (e'_m, t'_m)>$, where $t'_1<t'_2< \ldots<t'_m$, we define the *time interval* or *interval* of S' as $[t'_1, t'_m]$, and the *start time* and *end time* of S' as t'_1 and t'_m, respectively. Moreover, we say S' *occurs* at timestamp t'_m. S' is a *subsequence* of $\hat{S}$ if there exist m integers $i_1, i_2,\ldots, i_m$, such that $1\le i_1<i_2< \ldots<i_m$ and $\forall 1\le j\le m$, $e'_j = e_{i_j}$ and $t'_j = t_{i_j}$. Given an episode α with a time bound ω_α, the *episode occurrence* or *occurrence* of α over $\hat{S}$ is an event sequence S with time interval $[t_s, t_e]$ satisfying that: (1) S is a subsequence of $\hat{S}$; (2) the events corresponding to the vertices in α have an one-to-one mapping to the events in S; (3) the order in which the events occur in S is consistent with the temporal order $\prec$ specified in α ; and (4) $t_e - t_s \le \omega_\alpha$. Under the given one-to-one mapping, each vertex v in α corresponds to a unique instance of $\varepsilon(v)$ in S, which is called the *mapping instance* of v in S. Given any two occurrences of α, O_1 and O_2, with intervals $[t_{s1}, t_{e1}]$ and $[t_{s2}, t_{e2}]$, respectively, if $t_{s1}\le t_{s2}<t_{e2}\le t_{e1}$ and $t_{e2}-t_{s2}<t_{e1}-t_{s1}$, we say that $[t_{s2}, t_{e2}]$ is *enclosed* by $[t_{s1}, t_{e1}]$. For ease of presentation, we also say that O_2 is *enclosed* by O_1. The second condition excludes two occurrences in the same time interval from consideration. A *minimal occurrence* of α is an (episode) occurrence that is not contained by any other (episode) occurrence of α. Note that, with this definition, there can be multiple minimal occurrences in the same time interval.

Episode Rule: An *episode rule* ρ is a 5-tuple $(\alpha, \beta, \omega_\alpha, \omega_\rho, \chi)$. Here, α is the episode representing the predicate of ρ and β is the event representing the *consequent* of ρ. ω_α and ω_ρ stand for the predicate window and rule window, respectively. The interpretation of ρ is that if α has an occurrence O with interval $[t_s, t_e]$ $(t_e - t_s \le \omega_\alpha)$, β will occur during interval $(t_e, t_s + \omega_\rho)$ with probability χ. We call the interval $(t_e, t_s + \omega_\rho)$ the *predicted interval* of the occurrence O. The notations used in the remainder of this paper and their definitions are shown in Table 1.

Table 1. Notations and their definitions

Notations	Definitions
ρ	An episode rule
α	The predicate episode of ρ
β	The consequent of ρ
ω_α	The predicate window of ρ
ω_ρ	The rule window of ρ

2.2 Basic Concepts

In the following, we first introduce the properties of minimal occurrences, and then show that only the minimal occurrences are not redundant.

Property 1. If the occurrence of α O_1 with interval $[t_{s1}, t_{e1}]$ is not a minimal occurrence, there must be a minimal occurrence of α O_2 with interval $[t_{s2}, t_{e2}]$ such that the predicted interval of O_1 is enclosed by that of O_2.

Proof. By definition, since O_1 is not the minimal occurrence, there exist some occurrences of α enclosed by O_1. For each of them, similarly, if it is not the minimal occurrence, it must also enclose some other occurrences of α. In this way, we can finally find a minimal occurrence enclosed by O_1, say O_2. The predicted intervals of O_1 and O_2 are $(t_{e1}, t_{s1}+\omega_\rho)$ and $(t_{e2}, t_{s2}+\omega_\rho)$, respectively. Since O_2 is enclosed by O_1, the relation between their start times and end times must be in one of three cases: (1) $t_{s1}=t_{s2}$ and thus $t_{e2}<t_{e1}$, (2) $t_{s1}<t_{s2}$ and $t_{e1}=t_{e2}$, and (3) $t_{s1}<t_{s2}<t_{e2}<t_{e1}$. No matter which case is held for O_1 and O_2, the predicted interval of O_1 is enclosed by that of O_2.

By Property 1, any non-minimal occurrence can be ignored. Therefore, we conclude that only the minimal occurrences of α should be matched for the prediction of β. We summarize it in Lemma 1.

Lemma 1. Given the episode rule ρ, the union of predicted intervals for β derived from all the occurrences of α is exactly the same as the union of predicted intervals derived from only the minimal occurrences of α.

Intuitively, we may infer that all the minimal occurrences should be matched if each of them has a unique end time. Unfortunately, it is possible to have multiple minimal occurrences in the same time interval. Therefore, we need a unique way to select one from a set of minimal occurrences whose time intervals are all identical. Moreover, since all of them correspond to the same predicted interval, we also need an efficient way to guarantee that only the unique one of them is matched. We introduce the concept of *latest occurrence*, as defined below, which represents a unique form for one kind of minimal occurrences. Based on the concept, only the minimal occurrences in this form will be matched.

Definition 1. *Latest instance of an event*. Given a timestamp t and an event instance (X, t_1), where $t_1 \le t$, if there does not exist another event instance (X, t_2) such that $t_1 < t_2 \le t$, (X, t_1) is called the *latest instance* of X to t.

Definition 2. *Latest occurrence of an episode*. Given a timestamp t, the occurrence $<(\varepsilon(v_1), t_1), (\varepsilon(v_2), t_2), \ldots (\varepsilon(v_n), t_n)>$ of α, where $t_n \le t$, is called the *latest occurrence* of α to t if both of the following conditions hold:

(a) For every sink v_j of α, $(\varepsilon(v_j), t_j)$ is the latest instance of $\varepsilon(v_j)$ to t;

(b) For every non-sink vertex v_k of α, if $v_{k1}, v_{k2}, \ldots v_{km}$ are the direct successors of v_k, where $1 \le k < n$ and $t_k < t_{k1} < t_{k2} < \ldots < t_{km} \le t_n$, $(\varepsilon(v_k), t_k)$ is the latest instance of $\varepsilon(v_k)$ to t_{k1}.

In the following, we further explore the characteristics of the latest occurrence. Let Σ_t denote the set of all occurrences of α that occur at time t. By Definition 2, the latest occurrence of α to t, denoted as LO_t, is included in Σ_t. Moreover, the time interval of LO_t must be equal to or shorter than that of any other occurrence in Σ_t. In other words, any occurrence in Σ_t is not enclosed by LO_t. If Σ_t has one or more minimal occurrences, LO_t must be one of them. However, Σ_t is not always in this case and LO_t is not necessarily a minimal occurrence. We call the latest occurrence that is also a minimal occurrence the *min-latest occurrence*. In this paper, only the min-latest occurrences will be matched. Fortunately, the latest occurrences that are not minimal can be identified by checking whether they contain any of special event instances in the min-latest occurrence previously obtained, which are defined as follows.

Definition 3. *Rejected event instance set*. Given the episode α and the min-latest occurrence $<(\varepsilon(v_1), t_1), (\varepsilon(v_2), t_2), \ldots (\varepsilon(v_n), t_n)>$ previously obtained, the *rejected event instance set* is defined recursively as follows:

(a) $(\varepsilon(v_1), t_1)$ is a rejected event instance;

(b) Given that $(\varepsilon(v_i), t_i)$ is a rejected event instance, for every direct successors v_j of v_i, where $1 \le i < j \le n$, $(\varepsilon(v_j), t_j)$ is also a rejected event instance if there is no occurrence of $\varepsilon(v_i)$ in the time interval (t_i, t_j).

Given the rejected event instance set, the latest occurrence that contains a rejected event instance must have the same start time as the min-latest occurrence previously obtained. On the other hand, a latest occurrence that does not contain a reject event instance must be a minimal occurrence. In the following, we formulate these findings as Lemma 2 and Lemma 3, respectively.

Lemma 2. Given a min-latest occurrence $O=<(\varepsilon(v_1), t_1), (\varepsilon(v_2), t_2), \ldots (\varepsilon(v_n), t_n)>$, the latest occurrence $O' =<(\varepsilon(v'_1), t'_1), (\varepsilon(v'_2), t'_2), \ldots (\varepsilon(v'_n), t'_n)>$, where $t_n < t'_n$, is not a minimal occurrence if it contains a rejected event instance deduced from O.

Proof: Assume that O' contains a rejected event instance $(\varepsilon(v_j), t_j)$ in O. If $j>1$, by Definition 3(b), there must be a direct predecessor of v_j, say v_i, such that $(\varepsilon(v_i), t_i)$, where $1 \le i < j$, is also a rejected event instance in O. Moreover, there is no occurrence of $\varepsilon(v_i)$ in the time interval (t_i, t_j). By Definition 1, $(\varepsilon(v_i), t_i)$ is the latest instance of $\varepsilon(v_i)$ to t_j. Therefore, by Definition 2, O' also contains $(\varepsilon(v_i), t_i)$. If $i>1$, the same reasoning is applied to find a direct predecessor of v_i. Eventually we will find that O' contains $(\varepsilon(v_1), t_1)$. Since $t_n < t'_n$, O is enclosed by O'. As a result, O' is not a minimal occurrence.

Lemma 3. A latest occurrence of α is a minimal occurrence if it does not contain any rejected event instance.

Proof: Let $O=<(\varepsilon(v_1), t_1), (\varepsilon(v_2), t_2), \ldots (\varepsilon(v_n), t_n)>$ be the previously obtained min-latest occurrence, producing the rejected event instance set. We prove the contrapositive of the lemma. Suppose that the conclusion is false, i.e., the latest occurrence $O'=<(\varepsilon(v'_1), t'_1), (\varepsilon(v'_2), t'_2), \ldots (\varepsilon(v'_n), t'_n)>$ is not a minimal occurrence, where $t'_n > t_n$. Then, we need to show that O' contains a rejected event instance. According to the relationship between the start times of O and O', there are three cases to consider.

(1) $t'_1 = t_1$: Since $t'_n > t_n$ and $(\varepsilon(v_1), t_1)$ is a rejected event instance deduced from O, $\varepsilon(v'_1)=\varepsilon(v_1)$ and $(\varepsilon(v'_1), t'_1)$ is a rejected event instance.

(2) $t'_1 > t_1$: Since O' is not a minimal occurrence, by Property 1, there must be a minimal occurrence enclosed by O'. Moreover, the minimum occurrence enclosed by O' must be O because O is the min-latest one previously obtained. As a result, t'_1 should be smaller than or equal to t_1. This contradicts with the assumption $t'_1 > t_1$.

(3) $t'_1 < t_1$: (i) For every sink v_i of α, $(\varepsilon(v_i), t_i)$ is the latest instance of $\varepsilon(v_i)$ to t_n. If there are some instances of $\varepsilon(v_i)$ coming after t_n, one of them must be the latest instance of $\varepsilon(v_i)$ to t'_n, and is contained by O'. Otherwise, $(\varepsilon(v_i), t_i)$ is also the latest instance of $\varepsilon(v_i)$ to t'_n, and is contained by O'. Therefore, for the mapping instance of v_i in O', its occurring time must be larger than or equal to t_i (the occurring time of $(\varepsilon(v_i), t_i)$ in O). (ii) For each non-sink vertex v_j and its direct successors $\{v_{i_1}, v_{i_2}, \ldots v_{i_k}\}$, where $j < i_1 < i_2 \ldots < i_k$, $(\varepsilon(v_j), t_j)$ is the latest instance of v_j to t_{i_1}. If v_{i_1} is a sink vertex, according to step (i), the occurring time of the mapping instance of v_j in O' must be larger than or equal to t_j. On the other hand, if v_{i_1} is not a sink vertex, we can recursively apply the same step on the mapping occurrences for v_{i_1} and its direct successors in O'. Eventually, when returned from the recursion, we can prove that the

occurring times of the mapping occurrences for all the direct successors for v_j are larger that or equal to those in O. As a result, we have the same result for any non-sink vertex v_j. Therefore, the result $t'_1 \geq t_1$ contradicts with the assumption $t'_1 < t_1$.

Only case (1) should be considered and we have that the premise is false. Thus we have finished the proof.

To conclude this section, only the latest occurrences that do not contain any rejected event instance are the min-latest occurrences we are looking for. This is the basic of our approach and the correctness is guaranteed as long as our approach always returns such kind of occurrences.

3 The CBS-Tree (Circular Binary Search Tree) Approach

In this section, we describe our tree-based method. Subsection 3.1 introduces the proposed tree structure, CBS-Tree, and the tree maintenance. In subsection 3.2, we present how to use CBS-Tree to accelerate the min-latest occurrence retrieval for a given predicate episode.

3.1 CBS-Tree

The CBS-Tree, which is an index of recently arrived events over the stream, can achieve a fast retrieval of the min-latest occurrence of a given episode. Let L be the number of event instances recorded in the CBS-tree. We design CBS-Tree as a complete binary tree with L leaves, where each leaf corresponds to an event instance and each node is numbered from left to right in bottom-up fashion. As a result, the most recent L event instances are kept in the leaves, which are numbered from 1 to L. Moreover, the event instance with timestamp t is assigned to the leaf node numbered $((t - 1) \bmod L + 1)$. In addition, each internal node is associated with two sets, the *event set* R_E and the *timestamp set* R_T, to respectively keep the events and timestamps of all the event instances assigned to its descendants.

Let the event stream $\hat{S}$ be <(A, 1), (B, 2), (A, 3), (C, 4), (D, 5), (C, 6), (B, 7), (D, 8), …>. For example, Figure 4 shows the CBS-Tree with 7 leaves, corresponding to the event instances in time interval [1, 7] over $\hat{S}$. Each node in the tree is numbered as mentioned and the sets R_E and R_T of each internal node are depicted above the node. In the leaf layer, node 1 stands for the event instance (A, 1), node 2 represents the event instance (B, 2), and so forth. For the internal node 8, since it is the parent of nodes 1 and 2, its event set is {A, B} and timestamp set is {1, 2}.

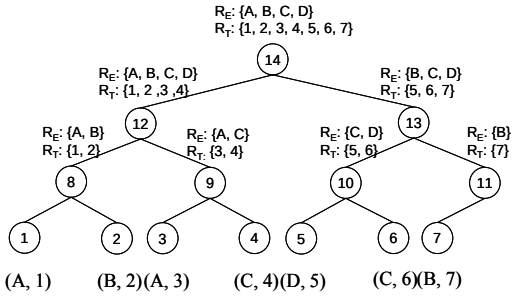


Figure 4: The CBS-Tree

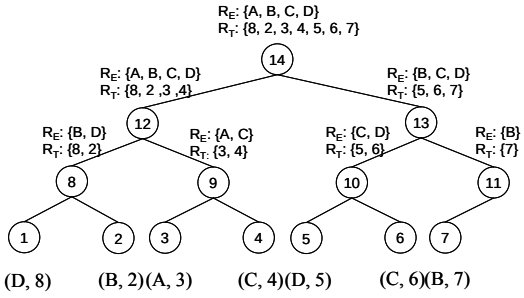


Figure 5: The CBS-Tree after updating

Tree Maintenance. Initially, since no event has arrived, the event set and timestamp set of each node are empty. Once an event arrives, the maintenance operation begins at the leaf node that corresponds to the timestamp of the arrived event, and then iteratively updates its ancestors in bottom-up fashion. For example, in Figure 5, we show the final state of CBS-Tree after the event instance (D, 8) arrives. Since we only need to record the events whose arrival times range from timestamps 2 to 8, node 1 corresponding to the event instance (A, 1) now corresponds to the event instance (C, 8) instead. Therefore, we update the event set and timestamp set of node 8 to {B, D} and {8, 2}, respectively. We then update node 12 and node 14 respectively. In this way, the tree maintenance at each timestamp requires traversing only one path from a leaf to the root. Thus, the time complexity of tree maintenance is O(logL).

3.2 CBS-Tree-Based Retrieval

We now present how to utilize the CBS-Tree for fast retrieval of the min-latest occurrences. According to Definition 2, we retrieve all the event instances of a latest occurrence in the inverse topological order of their corresponding vertices in the episode. Take Figure 2 as an example. We first find the mapping instances of vertex (III), then vertices (II) and (I). During the retrieval, if one of the mapping instances is a rejected event instance, we stop the retrieval and can be sure that there is no min-latest occurrence.

In Algorithm 1, we present the algorithm named *CBS-Tree Retrieval* (shortly CBR) for finding the min-latest occurrence of the episode α occurred within time interval [lt, rt], where rt is the current time and lt = rt – ω_α+ 1. In the algorithm, we first determine whether the newly arrived event can be the mapping instance of a sink in α. If this condition holds, we then retrieve the latest occurrence of α. Otherwise, there is no new min-latest occurrence of α with end time rt. To retrieve the min-latest occurrence, as depicted in lines 2-4 of Algorithm 1, we iteratively retrieve the latest instance mapped to every vertex in α by CBS-Tree, each of whose direct successors has been associated with a mapping instance in [lt, rt]. In this way, if there does not exist a mapping instance for a vertex, we say that no min-latest occurrence of α with end time rt exists (line 5). The procedure *FindLastestEventOccurrence* in line 4 returns the occurring time of the latest instance for a given event and a time interval. The left-endpoint of the specified time interval is always set to lt, and the right-endpoint is computed from the mapping instances of the direct successors of v by Definition 2 (line 3). FindLastestEventOccurrence will return −1 to indicate that the desired latest instance is not found or is a rejected event instance. Detailed descriptions of FindLastestEventOccurrence will be presented later. In line 6, once a new min-latest occurrence is found, the corresponding rejected event instance set will be derived.

Algorithm 1 *CBS-Tree Retrieval* (r, α, lt, rt, R)

Inputs: The root of the CBS-Tree r, the predicate episode α, the specified time interval [lt, rt], and the rejected event instance set R deduced from the min-latest occurrence of α last retrieved.

Output: The min-latest occurrence of α with end time rt.

1. **If** the event instance with arrival time rt does not correspond to a sink of α, return and report there does not exist the desired episode occurrence ;

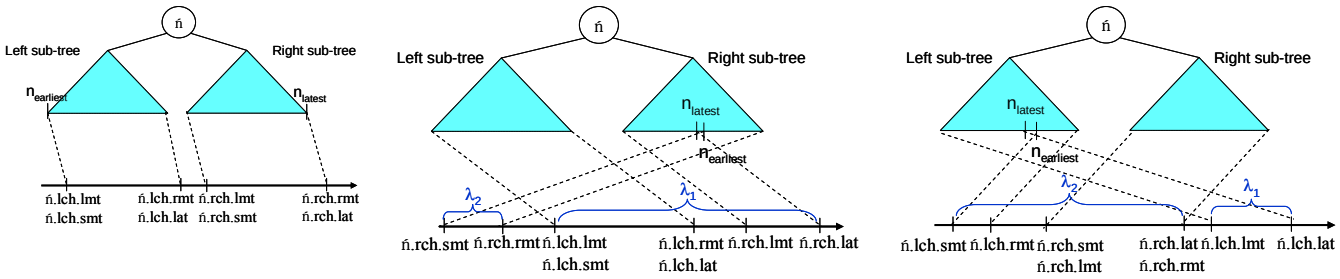
   Else let the event instance be the mapping instance of the sink corresponding to it ;
2. **While** there exists a vertex v of α, which does not map to an event instance and is a sink of α or all the direct successors of v have the mapping instances over the event stream do
3. { **If** v is a sink node t' = rt – 1 ;

Else let $\{v_1, v_2, \ldots v_i\}$ be the set of direct successors of v, $\{(\varepsilon(v_1), t_1), (\varepsilon(v_2), t_2), \ldots(\varepsilon(v_i), t_i)\}$ be the set of the mapping instances of $\{v_1, v_2, \ldots v_i\}$, and $t_1 < t_2 < \ldots < t_i$, set $t' = t_1 - 1$;

4. t = FindLastestEventOccurrence (v.event, lt, t', r) ; *// v.event: the event corresponding to vertex v; r: the root of the CBS-Tree*
5. **If** (t = –1) Return and report that there does not exist the min-latest occurrence with end time rt; }
6. Derive the rejected event instances from the newly retrieved min-latest occurrence by CBS-Tree, and report the retrieved min-latest occurrence ;

We now describe how the procedure FindLastestEventOccurrence proceeds by using CBS-Tree. In general, the retrieval for a latest instance begins at the root, and iteratively checks some children of the current node until a leaf node is reached. We will show that the total number of paths we have to check from the root to leaves is always at most 2. Hence the time complexity for retrieving an event instance is O(logL) with multiplier less than or equal to 2.

In the following, we will focus on the general case of CBS-Tree in which every leaf corresponds to an event instance. That is, at least L event instances have been received by CBS-Tree. At the end of this section, we then discuss the special case of CBS-Tree where some leaves do not correspond to event instances. For each node ń, we use ń.id, ń.lch, ń.rch, ń.LR_E and ń.RR_E to denote the node ID, the left child, the right child, the event sets of the left child and the right child, respectively. Let ń.lmt and ń.rmt be the timestamps corresponding to the event instances represented by the leaf nodes under ń whose ID number is the smallest (the left-most leaf node) and the largest (the right-most leaf node), respectively. Moreover, we use ń.lat and ń.smt to denote the largest and smallest timestamps in the timestamp set of node ń, respectively. The leaf nodes corresponding to them are denoted as n_{latest} and $n_{earliest}$, respectively. According to the location of n_{latest} under ń, we have three types of situations to consider. As shown in Figure 6, Type 1 means that n_{latest} is the rightmost leaf node, while the other two types indicate that n_{latest} is under either ń.lch or ń.rch. The timestamps are also depicted in the figures.



Type 1: n_{latest} is the rightmost leaf

Type 2: n_{latest} is under ń.lch

Type 3: n_{latest} is under ń.rch but not the rightmost leaf

Figure 6: Three types of situations

Two useful properties for event instance retrieval are explored from the relation of the timestamps represented by the leaf nodes under a given node. We respectively present them in Property 4 and Property 5.

Property 4. Given a CBS-Tree and a node ń in it, if and only if ń.lch.lat < ń.rch.smt, among the leaf nodes under ń, the latest updated one (n_{latest}) must be the right-most leaf node.

Proof. CBS-Tree circularly updates the leaf nodes from left to right for inserting the event instances in the order of timestamps. Therefore, every leaf node except the latest updated one is associated with a timestamp smaller than the timestamp of its right sibling. Assume that the latest updated leaf X is not the right-most one. X can be either the other leaf under ń.rch or a leaf under ń.lch. In the former case, the right sibling of X is associated with the smallest timestamp represented by ń.rch and ń. Thus, ń.rch.smt is equal to ń.smt. Since the timestamp associated with any leaf under ń.lch is larger than the smallest timestamp, ń.lch.lat > ń.rch.smt. In the latter case, X is associated with the largest timestamp represented by ń.lch and ń. Thus, ń.lch.lat is equal to ń.lat. Since the timestamp associated with any leaf under ń.rch is smaller than the largest timestamp, ń.rch.smt < ń.lch.lat. Because in both cases the condition ń.lch.lat < ń.rch.smt does not hold, we have proved that the latest updated leaf node is the right-most one if ń.lch.lat < ń.rch.smt. On the other hand, since the leaf nodes under ń are updated from left to right, if X is the right-most leaf under ń.rch, any leaf node under ń.rch must be updated later than any other under ń.lch. Therefore, we have ń.lch.lat < ń.rch.smt. The proof is thus completed.

Based on the above property, the following corollary can be derived.

Corollary 1. Given a CBS-Tree and a node ń in it, if ń.lch.lat < ń.rch.smt, (1) for any two leaf nodes under ń l_1 and l_2, if and only if l_1.id < l_2.id, the timestamp of l_1 < that of l_2, and (2) ń.lch.smt = ń.lch.lmt, ń.lch.lat = ń.lch.rmt, ń.rch.smt = ń.rch.lmt, and ń.rch.lat = ń.rch.rmt.

The latest event instance retrieval can be categorized into four cases, where the first three are treated based on Property 4 while the last one is treated based on Property 5, which will be presented later. The first three cases occur when ń.lch.lat < ń.rch.smt, and by Property 4, only Type 1 in Figure 6 should be considered. Given a query event ė, the specified time interval [lt, rt], and the currently retrieved node ń, the largest value of rt and the smallest value of lt are limited by ń.rch.rmt and ń.lch.lmt, respectively. We therefore divide the specifications of lt and rt into three cases: (1) rt=ń.rch.rmt, (2) lt = ń.lch.lmt and rt < ń.rch.rmt, and (3) lt> ń.lch.lmt and rt < ń.rch.rmt. Figure 7 illustrates these cases and their sub-cases. In each of the cases, if [lt, rt] has an overlap with the time interval of ń.rch, we should first consider ń.rch and then ń.lch. In the following, the pseudo codes for each case are presented below its descriptions. In the cases, we use t to denote the timestamp returned to the procedure FindLastestEventOccurrence and initialize it as −1.

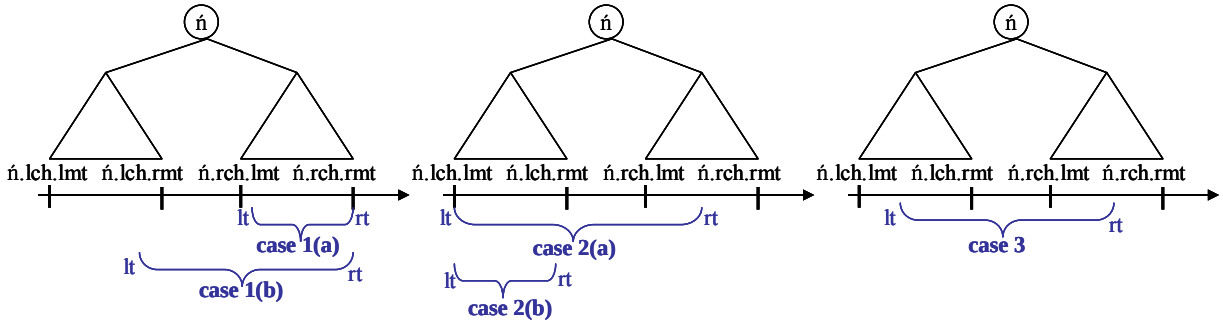


Figure 7: Illustration of the first three cases

Case 1. rt = ń.rch.rmt. There are two sub-cases. (a) The specified time interval [lt, rt] is entirely covered by the time interval [ń.rch.lmt, ń.rch.rmt], i.e., lt ≥ ń.rch.lmt. By the definition of latest event instance, if there is an answer, it must be found in the leaf nodes under ń.rch. Therefore, as shown in line 1 of Procedure Case_1, if ė is

contained in the event set of ń.rch, we then search the right sub-tree. Note that, we pass the larger of lt and ń.rch.lmt as the left bound of the specified time interval for searching the right sub-tree. (b) The specified time interval [lt, rt] has an overlap with the time interval [ń.lch.lmt, ń.lch.rmt], i.e., lt ≤ ń.lch.rmt. We first check whether there is an instance of ė in the right sub-tree, i.e., check whether ė is contained in the event set of ń.rch. If so, we can obtain the answer by searching the right sub-tree. Otherwise, we check if there exists any instance of ė in the left sub-tree. As shown in line 2 of Case_1, we search the left sub-tree by passing the time interval [lt, ń.lch.rmt]. To sum up for case 1, only one of the two sub-trees should be searched. Therefore, we can make a unique choice on the two children to follow during the retrieval. Moreover, the same procedure of Case 1 will be repeated in the next tree level. As a result, at most 1 path of CBS-tree should be traversed in case 1.

Procedure *Case_1*

1. **If** (ė ∈ ń.RR_E) t = FindLatestEventOccurrence (ė, max{lt, ń.rch.lmt}, rt, ń.rch) // *sub-cases (a) and (b)*
2. **Else if** (ė ∈ ń.LR_E and lt ≤ ń.lch.rmt)
   t = FindLatestEventOccurrence (ė, lt, ń.lch.rmt, ń.lch) ; // *sub-case (b)*

Case 2. rt < ń.rch.rmt and lt = ń.lch.lmt. There are also two sub-cases. (a) The specified time interval [lt, rt] has an overlap with the time interval [ń.rch.lmt, ń.rch.rmt], i.e., rt ≥ ń.rch.lmt. Two steps are needed for this case. At step (i), we check whether there is an instance of ė within [ń.rch.lmt, rt] (line 1 of Case_2). If not, we then proceed to step (ii), i.e., searching the left sub-tree (line 2 of Case_2). (b) The specified time interval [lt, rt] is entirely covered by the time interval [ń.lch.lmt, ń.lch.rmt], i.e., rt ≤ ń.lch.rmt. Obviously, if there is an answer, it must be found in the left sub-tree. Line 2 of Case_2 also corresponds to this sub-case. Note that, line 2 integrates step (ii) of sub-case (a) and sub-case (b), and we pass the smaller of rt and ń.lch.rmt as the right bound of the specified time interval for searching the left sub-tree. Moreover, it will fall into case 2 again in the next tree level, when line 1 is executed, and fall into case 2 or case 1, when line 2 is executed. To sum up, at most 2 paths will be traversed if the answer exists. One is to repeatedly apply case 2 in the deeper tree levels, and then finally ė is not found in the leaves, while the other is for the execution of line 2 (case 2 turns to case 1), and the answer is found.

Procedure *Case_2*

1. **If** (rt ≥ ń.rch.lmt and ė ∈ ń.RR_E) t = FindLatestEventOccurrence (ė, ń.rch.lmt, rt, ń.rch) ; //*step (i) of sub-case (a)*
2. **If (**ė ∈ ń.LR_E and t = –1)
   t = FindLatestEventOccurrence (ė, lt, min{rt, ń.lch.rmt}, ń.lch) ; // *step (ii) of sub-case (a) and sub-case (b)*

Case 3. rt < ń.rch.rmt and lt > ń.lch.lmt. This case can be reduced to case 1 or case 2, or recursively perform by itself. For this case, we first check the right sub-tree (line 1 of Case_3), and then the left sub-tree (line 2 of Case_3) if no answer exists in the right sub-tree. As ń.rch.lmt is passed in line 1, case 3 will be reduced to case 2 in the next tree level. Otherwise, case 3 will be reconsidered again. Similarly, as ń.lch.rmt is passed in line 2, case 3 will be reduced to case 1 in the next tree level. Otherwise, case 3 will be reconsidered again. In this way, if the answer exists in the right sub-tree, we can obtain it by traversing at most 2 paths according to case 2. Otherwise, it costs at most 1 path to determine if there is no answer in the right sub-tree, and at most 1 path to retrieve the left sub-tree according to case 1. In total, case 3 needs at most 2 paths to obtain the answer.

Procedure *Case_3*

1. **If** ($\dot{e} \in \acute{n}.RR_E$ and $rt \geq \acute{n}.rch.lmt$)
   t = FindLatestEventOccurrence ($\dot{e}$, max{lt, ń.rch.lmt}, rt, ń.rch) ; // *case 2 or case 3 itself*
2. **If** ($\dot{e} \in \acute{n}.LR_E$, $lt \leq \acute{n}.lch.rmt$, and $t = -1$)
   t = FindLatestEventOccurrence ($\dot{e}$, lt, min{rt, ń.lch.rmt}, ń.lch) ; // *case 1 or case 3 itself*

We now present Property 5 followed by Case 4.

Property 5. Given a CBS-Tree and a node ń in it, let n_{latest} be the latest updated one among the leaves under ń. For any two leaf nodes under ń n_x and n_y, the timestamp of n_x is larger than that of n_y if $n_x.id \leq n_{latest}.id$ and $n_y.id > n_{latest}.id$.

Proof. If n_{latest} is the right-most one among the leaves under ń, the condition $n_y.id > n_{latest}.id$ never hold. Therefore, we can divide the leaves under ń into two non-empty sets λ_1 and λ_2 such that $n_x.id \leq n_{latest}.id$ $\forall n_x \in \lambda_1$ and $n_y.id > n_{latest}.id$ $\forall n_y \in \lambda_2$. Since the leaves in CBS-Tree are circularly updated from left to right, for any two leaves $n_x \in \lambda_1$ and $n_y \in \lambda_2$, the timestamp of n_x is larger than that of n_y.

According to Property 4, the following corollary can be derived.

Corollary 2. Given a CBS-Tree and a node ń in it, for two leaf nodes under ń n_x and n_y, where $n_x.id \leq n_{latest}.id$ and $n_y.id \leq n_{latest}.id$, the timestamp of n_x is smaller than that of n_y if and only if $n_x.id < n_y.id$. The same statement also applies to the case where $n_x.id > n_{latest}.id$ and $n_y.id > n_{latest}.id$.

The last case occurs when ń.lch.lat > ń.rch.smt, and by Property 4, either Type 2 or Type 3 in Figure 6 should be considered. In Type 2, n_{latest} is under ń.lch and the time interval of the left sub-tree can be thus divided into two regions [ń.lch.lmt, ń.lch.lat] and [ń.lch.smt, ń.lch.rmt]. Moreover, following the proof of Property 5, the two sets of leaves λ_1 and λ_2 are associated with timestamps in the intervals [ń.lch.lmt, ń.lch.lat] and [ń.lch.smt, ń.rch.rmt], respectively. As shown in Figure 6, to find the latest instance of an event, we should first consider λ_1 and then λ_2. In Type 3, n_{latest} is a leaf node under the right sub-tree but not the right-most one. Similarly, the time interval of the right sub-tree is divided into two regions [ń.rch.lmt, ń.rch.lat] and [ń.rch.smt, ń.rch.rmt]. Moreover, in this situation, λ_1 and λ_2 correspond to the interval [ń.lch.lmt, ń.rch.lat] and [ń.rch.smt, ń.rch.rmt], respectively.

Case 4. There are two sub-cases to consider in this case. (a) $rt \geq$ ń.lch.lmt. It means that rt is located at the interval of λ_1 for either Type 2 or Type 3, as shown in Figure 8, and thus the interval [lt, rt] may span the intervals of both sets λ_1 and λ_2. Therefore, we first search λ_1 in the interval [ń.lch.lmt, rt] and then λ_2 in [lt, ń.rch.rmt]. For the retrieval in [ń.lch.lmt, rt], if lt > ń.lch.lmt, [lt, rt] is entirely covered by [ń.lch.lmt, rt], which can be regarded as Case 3. Otherwise, the retrieval in [ń.lch.lmt, rt] belongs to Case 2 since the left bound is fixed to ń.lch.lmt. Therefore, it requires at most two paths to find the answer or one path to recognize no answer in [ń.lch.lmt, rt]. For the later, the interval [lt, ń.rch.rmt] (λ_2) is then considered. Similarly, the retrieval on interval [lt, ń.rch.rmt] belongs to Case 1 since the right bound is fixed to ń.rch.rmt. Therefore, it only requires one path for the retrieval. (b) $rt <$ ń.lch.lmt. Since the leaf node corresponding to ń.lch.lmt is the leftmost one in the set λ_1 as defined in the proof of Property 5, any leaf node in the interval [lt, rt] must be in the set λ_2. Depending on whether rt is equal to ń.lch.lmt or not, this sub-case can be regarded as Case 1 (rt = ń.lch.lmt) or Case 3 (rt < ń.lch.lmt), respectively.

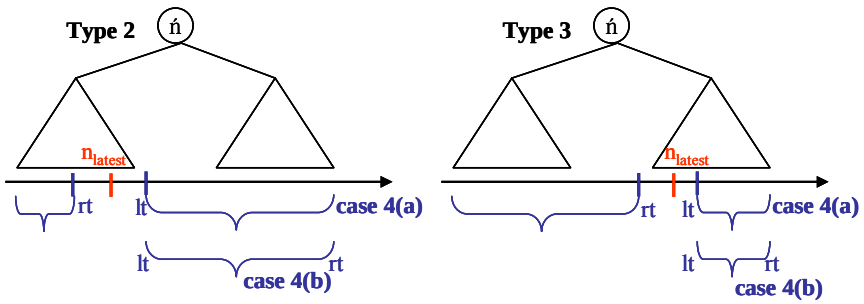


Figure 8: The sub-cases of Case 4

In procedure Case_4, lines 1-4 stand for the sub-case (a) while 5-6 correspond to the sub-case (b). In line 1, if the condition ń.rch.lmt ≠ ń.rch.smt holds, it implies that n_{latest} is under the right sub-tree, which corresponds to Type 3 in Figure 6. Therefore, we should first consider whether the answer exists in the right sub-tree. In line 2, we then consider the left sub-tree. Note that, the predicate lt ≤ ń.lch.rmt is used to exclude the case that both lt and rt all fall in the right sub-tree. If there is no answer found from λ_1, we then consider λ_2, and the corresponding codes are in lines 3-4. In line 3, the predicate ń.lch.lmt – 1 = ń.rch.rmt ensures that the leaves can be retrieved in correct order. For example, consider Figure 9 and the query parameters: ė=C, lt=2, and rt=8. The correct answer should be timestamp 6. We first traverse nodes 14 and 12. If we ignore the predicate ń.lch.lmt – 1 = ń.rch.rmt, we will then subsequently retrieve nodes 9 and then 4, and finally return timestamp 4 as the answer! With the predicate ń.lch.lmt – 1 = ń.rch.rmt, the first time to call the procedure FindLatestEventOccurrence in line 3 is at the time point returned from node 12. As a result, node 6 will be retrieved first then and node 4. In lines 5-6, we first consider the right sub-tree (line 5) and then the left sub-tree (line 6).

Procedure *Case_4*

1. **If** (rt ≥ ń.lch.lmt) // *sub-case (a)*
   If (ė ∈ ń.RE, ń.rch.lmt ≠ ń.rch.smt, and rt ≥ ń.rch.lmt)
   t = FindLatestEventOccurrence (ė, max{lt, ń.rch.lmt}, rt, ń.rch) ; // *case 2 or case 3*
2. **If** (t = –1, ė ∈ ń.LE, and lt ≤ ń.lch.rmt)
   t = FindLatestEventOccurrence (ė, lt, min{ń.lch.lat, rt}, ń.lch) ; // *case 1, case 2 or case 3*
3. **If** (t = –1, and ń.lch.lmt – 1 = ń.rch.rmt)
   If (ė ∈ ń.RE and lt ≤ ń.rch.rmt)
   t = FindLatestEventOccurrence (ė, max{ń.rch.smt, lt}, ń.rch.rmt, ń.rch) ; // *case 1*
4. **If** (t = –1, ė ∈ ń.LE, and lt ≤ ń.lch.rmt)
   t = FindLatestEventOccurrence (ė, lt, ń.lch.rmt, ń.lch) ; // *case 1*
   Else return
5. **Else** // *sub-case (b)*
   If (ė ∈ ń.RE and rt ≥ ń.rch.smt)

t = FindLatestEventOccurrence (ė, max{ń.rch.smt , lt}, rt, ń.rch) ; // *case 1, case 2, or case 3*

6. **If** (t = –1, ė ∈ ń.LE, and lt < ń.rch.smt)

   t = FindLatestEventOccurrence (ė, lt, min{ń.lch.rmt, rt}, ń.lch) ; // *case 1 or case 3*

In the following, we present the algorithm FindLastestEventOccurrence for the latest event instance retrieval. Given the event ė, the specified time interval [lt, rt], and a node of the CBS-Tree ń, the algorithm considers one of the four cases presented above in lines 3-7. In lines 1-2, whenever an instance in a leaf is retrieved, we check if it is a rejected event instance. If so, we terminate the retrieval; otherwise, the occurring time of the instance is returned.

Algorithm 2 *FindLatestEventOccurrence* (ė, lt, rt, ń)

Inputs: The queried event ė, the specified time interval [lt, rt], and the currently retrieved tree node ń

Output: The occurring time of the latest occurrence of ė in time interval [lt, rt]

1. **If** (ń is a leaf node)
2. **If** the event instance of ń is a rejected event instance, return –1 to the calling procedure CBR ;

   Else return the occurring time of the event instance represented by ń to the calling procedure CBR ;
3. **If** (ń.rch.smt > ń.lch.lat) // *cases 1, 2, and 3*
4. **If** (rt = ń.rch.rmt)

   Invoke Case_1 ;
5. **Else if** (lt = ń.lch.lmt)

   Invoke Case_2 ;
6. **Else**

   Invoke Case_3 ;
7. **Else** // *case 4*

   Invoke Case_4 ;

Example 1. We illustrate the retrieval of event B within time interval [2, 8] on the CBS-Tree in Figure 9. Initially, at the root, since the largest timestamp represented by node 12 (i.e. 8) is larger than the smallest timestamp represented by node 13 (i.e. 5), we retrieve the answer by case 4. Next, since rt = 8, we first retrieve the answer within time interval [8, 8] from the left sub-tree of the root according to the first sub-case of case 4. However, we discover that there is no answer in the left sub-tree since event B is not contained in the event set of node 8. Therefore, we retrieve the right sub-tree of the root. Since event B is in the event set of node 13, we conclude that the latest instance of ė must appear in the time interval [5, 7]. This case belongs to case 1, and it takes us only one path to find the answer on the leaf node 7. To sum up, during the retrieval, our approach traverses one path, 14→12→8, to realize that no answer exists in time interval [8, 8]. Then, it traverses one more path, 14→13→11→7, to retrieve the event instance (D, 7).

Let N be the number of vertices in an episode. Since each event instance retrieval on CBS-Tree takes at most 2 paths, the time complexity for the min-latest occurrence matching procedure at each timestamp is N·logL, where L is the number of leaf nodes in the tree. Moreover, if a min-latest occurrence is retrieved, the time complexity for deriving the rejected event instances is also N·logL. Finally, it requires logL time complexity for

the tree update at each timestamp. As a result, the time complexity of CBS-Tree will be $2 \cdot N \cdot \log L + \log L = O(N \log L)$.

For the special case that the number of received event instances is smaller than L, we adopt the *DirectMatch* [4] which simply retrieves the min-latest occurrence of a predicate episode from raw data. Note that, the CBR algorithm can be extended to handle such special cases.

The advantages of utilizing CBS-Tree for the min-latest occurrence matching are: (1) since the number of events coming in a time window is usually very large, CBS-Tree provides an efficient way to retrieve the desired instances from the window, (2) the space required by CBS-Tree is constant to the number of vertices in a given predicate episodes. This is especially effective for memory management when the scale of given predicate episodes is large.

4 Experiment Results

In this section, we evaluate the performances of CBS-Tree and ToFel by a series of experiments on real datasets. All the experiments are performed on a Pentium IV 2.4GHz PC with 2GB RAM and under the Microsoft Windows XP environment. We consider two real datasets for performance evaluation, which are respectively described as follows.

Intel lab data. We download the first real dataset from "http://db.csail.mit.edu/labdata/labdata.html," which was obtained from 54 wireless sensors deployed in the Intel Berkeley Research lab during 2/28/2004 - 4/5/2004. Each record of the dataset is a reading collected from a sensor and consists of the information of humidity, temperature, light, and the other attributes at a time. In the experiments, we only focus on the sensor ID, humidity, and temperature. An event then maps to a unique combination of a sensor ID, a unit percentage humidity value, and an integer temperature value. As a result, there are totally 3,931 different events in the transformed dataset. We also remove the records with abnormal data values and then, there are totally 938,138 records remained in the dataset. In addition, the data records are sorted in ascending order of their generation time and each of them is assigned to a unique record number. For simplify, we denote the record number as the arrival time of the record instead of its generation time.

Traffic data. We download the second real dataset from "http://www.d.umn.edu/~tkwon/TMCdata/TMCarchive.html," which was collected by the Traffic Management Center (TMC) of US from the loop detector with number 10 located in the Twin Cities Metro freeways. Each record of the data consists of the flow and occupancy of the section sensed by the detector at a 30-second interval in the days during 1/1/2006 - 6/30/2006. A unique value of flow (integer) and a unique unit percentage for occupancy are mapped into a specific event. As consequent, there are totally different 295 events in the dataset and the dataset contains 789,120 records.

All the notations used for generating predicate episodes and their definitions are stated as follows. *AveV*: the average number of vertices in a predicate episode, *AveE*: the average number of out-edges for a vertex in the predicate episodes, and *AveW*: the average size of the predicate window in a predicate episode. Let $\Gamma(\delta)$ be the function of normal distribution with mean δ. To generate a predicate episode α from the real dataset, we first generate the corresponding predicate window ω_α by Γ(AveWindow). Then, we randomly pick Γ(AveVertex) events as vertices from the records with record numbers t, t+1, t+2, …, t+AveWindow−1 in the dataset. Let the vertices in α be $v_1, v_2, \ldots v_n$. For each vertex v_k $(1 \le k \le n)$, we compute the number of its out-edges n_{edge} from Γ(AveE). Then, we randomly select n_{edge} different vertices from the set $\{v_i | k<i\}$ as its direct successors. To

avoid a vertex v_j ($1<j\leq n$) being over-connected, we also add AveE dummy vertices denoted as v_{n+1}, v_{n+2}, …, v_{n+AveE} during the edge construction. If v_k selects one of these vertices to be its direct successor, no operation is carried out for this connection. If there are many isolated sub-graphs in a seed, we add the minimum number of edges to combine them into a connected graph. Note that, we only show the performance of our approach on the matching of the min-latest occurrences, since it is the main cost of the event prediction problem. Therefore, we only need to generate the set of predicate episodes with predicate windows instead of the episode rules.

We set various parameters to evaluate our efficiency with respect to the sizes of predicate windows, the structure of episodes, and the number of episodes. If not specified in the experiments, the default settings for the number of episodes, AveV, and AveE are 10,000, 5, and 2, respectively. In the following, each figure corresponds to the results from a set of experiments. Moreover, each point of a curve in the figure stands for the execution time of one approach.

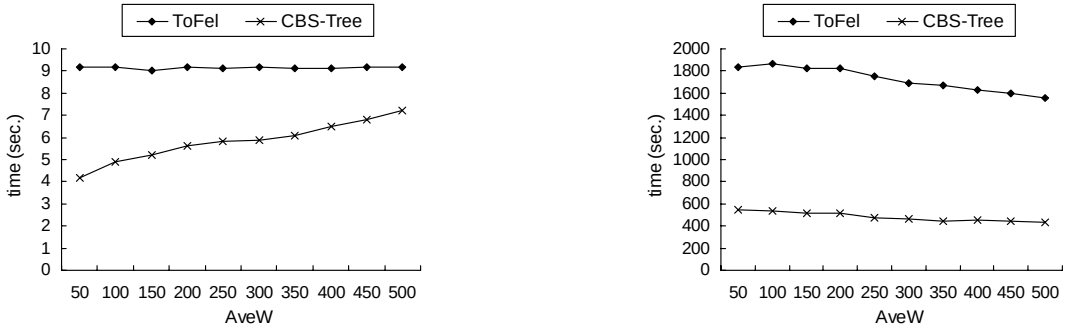
At first, we are interested in how the data distribution and the size of predicate window influence the performances of CBS-Tree and ToFel. Given the predicate episode α, if the event instances corresponding to the vertices of α come often, ToFel needs to frequently maintain the queues of α for the arrived event instances even if they finally do not form the min-latest occurrences. On the other hand, since the matching of min-latest occurrence in CBS-Tree is triggered only by the event instances corresponding to the sinks of α, the above situation which causes ToFel to frequently perform queue updates is not significant to CBS-Tree. Therefore, ToFel is more sensitive than CBS-Tree when the stream data is relatively dense. Table 2 shows the total frequencies of the k events with highest frequencies in the two real datasets, respectively. Intuitively, a dataset with a smaller k and a larger total frequency indicates a denser dataset. As we can see, the density of Traffic dataset is significantly higher than Intel lab dataset.

For a sparser dataset, as the size of predicate window becomes larger, the event instances are more likely to become a part of the minimal occurrence and the number of answers increases. Therefore, the execution times of the two approaches will grow with the increase of AveW since there are more episode occurrences in a large window (this also implies more minimal occurrences). For a denser dataset, events can form episode occurrences in a small window. However, in a large window, since some occurrences turn to redundant ones, the number of minimal occurrences is thus reduced. As a result, the execution times of the two approaches will decrease with the growth of AveW. The figures shown in Figure 9 depict the performance of the two approaches with respect to the AveW values varying from 50 to 500 on Intel lab data and the traffic data, respectively. CBS-Tree outperforms ToFel on all the cases in both figures. In Figure 9(a), the execution time of CBS-Tree slightly increases with the growth of window size, but is still better than that of ToFel in the cases of lager window sizes. In Figure 9(b), since the traffic dataset is relatively denser than Intel lab dataset, ToFel spends significantly more time to process the traffic data against the Intel lab data. In the following, we evaluate the performances of CBS-Tree and ToFel by three sets of experiments with different settings of AveV, AveE, and the number of predicate episodes, respectively. Due to the lack of space, we will only show the cases that set AveW to 500.

Table 2. Event frequencies in the real datasets

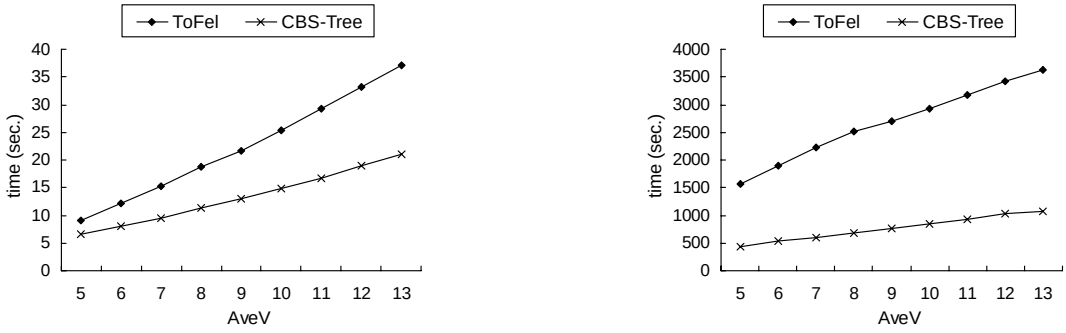
	k=5	k=10	k=15	k=20	k=25	k=30	k=35	k=40
Intel lab dataset	1.3%	2.6%	3.8%	4.8%	5.8%	6.7%	7.6%	8.4%
Traffic dataset	26.4%	37.3%	44.2%	49.5%	53.4%	56.2%	58.6%	60.6%

The execution times of the two approaches with various AveV values are shown in Figure 10. As the number of vertices in an episode increases, the two approaches need more time to match a min-latest occurrence, and the number of answers decreases as the AveV value increases. CBS-Tree outperforms ToFel in either Intel lab data or the traffic data. In the experiments, most of the irrelevant event instances can be efficiently skipped by CBS-Tree when the value of AveV is large. Moreover, CBSTree exhibits an excellent performance when the dataset is relatively dense.



(a) Intel lab data (b) Traffic data

Figure 9: Execution time with different AveW



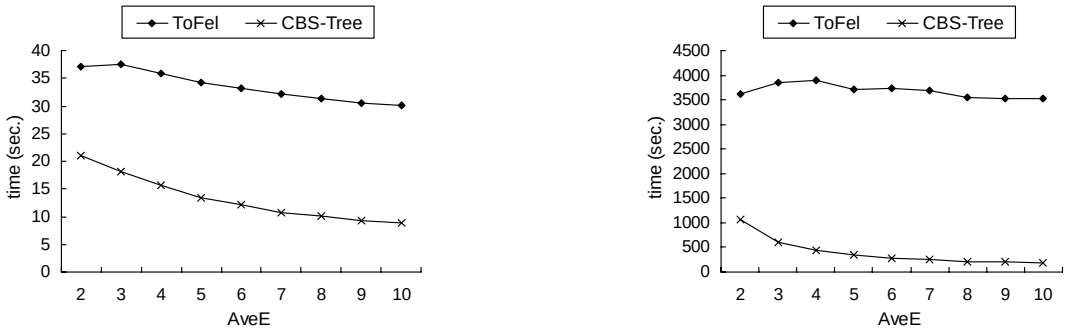
(a) Intel lab data (b) Traffic data

Figure 10: Execution time with different AveV

In Figure 11, we evaluate the performances of the two approaches with respect to different settings of AveE. In this experiment, we set AveV to 14 for all the cases. In general, the larger the number of edges in an episode is, the stronger the temporal constraints between vertices in the episode will be. This results in a smaller number of min-latest occurrences in the answer set. Therefore, the running time of the approaches decreases as the AveE

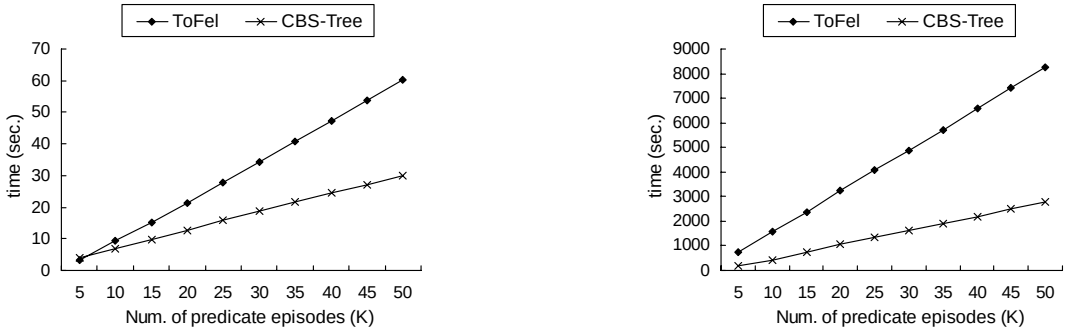
value increases. In the figures, CBS-Tree exhibits better performance than ToFel no matter how much the value of AveE is. For ToFel, when the number of edges in an episode is large, ToFel will take more time to maintain the queue (In ToFel, an event instance for a vertex can be kept if there are some event instances kept for all its direct predecessors. Intuitively, there are more direct predecessors for a vertex when the AveE value is larger). This cancels out the time saved when fewer answers are found under a large value of AveE.

Figure 12 shows the experimental results with respect to different numbers of predicate episodes. The execution times of the two approaches increase linearly with the increasing number of episodes. CBS-Tree has better performance than ToFel except when the number of predicate episodes is 5,000 in Figure 12(a).



(a) Intel lab data (b) Traffic data

Figure 11: Execution time with different AveE



(a) Intel lab data (b) Traffic data

Figure 12: Execution time with different numbers of predicate episodes

5 Related Work

In the following, we first briefly introduce the works and then discuss the difference between them and ours. The *ECA-Rules* (Event-Condition-Action-Rules) in active database management systems are used to trigger a timely response when the conditions of corresponding predicates are satisfied. Generally, the predicate of an ECA-Rule is represented as a *composite* (*complex*) *event*. Over the past few decades, significant projects such as Ode [8] and SAMOS [7] have developed individual active database management systems. One of the core issues in these projects is to propose the algebra with a rich number of event operators to support various types of queries (composite events). In addition, SAMOS and Ode implement their composite event processors based on colored petri nets and finite automata, respectively. In content-based subscription systems [1, 5], senders simply publish events (messages or news) associated with a set of predefined attribute-value pairs to the receivers, while receivers subscribe the interested contents of events by specifying individual predicates composed of a set of attribute-value-operator pairs to filter out uninterested events. Aguilera et al. [1] indexes subscriptions as a tree structure where a path from the root to a leaf corresponds to a set of subscriptions whose sets of attribute-value-operator triples are identical. An event will be evaluated by traversing the tree from the root to some leaves. If a leaf is finally reached, the event will fit the subscriptions corresponding to the retrieved path. Demers et al. [5] allow users to specify a subscription whose match is formed from a set of events. They implement their system based on a nondeterministic finite state automata cooperated with indexing structures. In this way, the system can deal with multiple descriptions simultaneously through the proposed indexing techniques on the automata. Wu et al. [14] proposed an event sequence query model (*SASE*) with attribute value comparison for RFID data streams. A query plan based on NFA with a stack structure is proposed to retrieve the query occurrences from the stream. In the RFID-based data stream management systems proposed in [6, 13], query models and system architectures were discussed but no detailed implementations are provided. In addition, there are relational operator-based event processors proposed [2, 3] over data streams in recent years. Most of them aim at optimizing the query plan (an order of join steps for matching queries) to fit the specified resource constraints, such as CPU time and memory space.

The difference between these works and ours mainly lies on our concept of the minimum occurrence and the corresponding matching process over event streams. The algorithms based on automata [5, 8, 14] or petri nets [7] cannot avoid redundant predictions. Some of the complex event processors [5, 14] cannot cope with graph-based queries and some [2, 3] lack a sequence operator. The query format considered in [1] is set-based while ours is graph-based, and their answer must be matched at a timestamp while ours is matched over a window of timestamps. As a result, all of them are not adaptable to handle episode-based queries.

6 Conclusion and Future Work

In this paper, we propose a novel approach to match the predicate episode of an episode rule over event streams for the prediction of the consequent event. Our approach only finds the minimal occurrence such that the duplicate prediction can be avoided. The approach is based on the CBS-Tree which maintains the recent event instances such that the minimal occurrence can be efficiently retrieved. We formulate four cases for retrieving the event instances of the minimal occurrence via CBS-Tree. Each of the four cases only takes O(logL) time complexity to retrieve the desired event instance, where L is the number of event instances indexed in the tree. We evaluate the performance of our approach by varying the size of predicate window, the structural scale of the episode such as the number of vertices, and the number of predicate episodes. The experiment results show that our approach outperforms the previous work in most cases. The results also reveal that CBS-Tree has the

outstanding performance when the dataset is relatively dense. In the future, we will extend our approach to handle more complex predicates such as the episodes with negation operators or the events with multiple attributes.

Reference

1. Abadi, D.J. et al., "Aurora: A Data Stream Management System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 666, 2003.
2. Ayad, A. and J.F. Naughton, "Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 419-430, 2004.
3. Babu, S., R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive Ordering of Pipelined Stream Filters," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 407-418, 2004.
4. Cho, C.W., Y. Zheng, and A.L.P. Chen, "Continuously Matching Episode Rules for Predicting Future Events over Event Streams," *Proceedings of joint conference of Asia-Pacific Web Conference and International Conference on Web-Age Information Management*, 884-891, 2007.
5. Demers, A.J., J. Gehrke, M.S. Hong, M. Riedewald, and W.M. White, "Towards Expressive Publish/Subscribe Systems," *Proceedings of International Conference on Extending Database Technology*, 627-644, 2006.
6. Franklin, M.J., S.R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, "Design Considerations for High Fan-In Systems: The HiFi Approach," *Proceedings of Biennial Conference on Innovative Data Systems Research*, 290-304, 2005.
7. Gatziu, S. and K.R. Dittrich, "SAMOS: an Active Object-Oriented Database System," *IEEE Database Engineering Bulletin*, 15(1-4), 23-26, 1992.
8. Gehani, N.H., H.V. Jagadish, and O. Shmueli, "Composite Event Specification in Active Databases: Model & Implementation," *Proceedings of International Conference on Very Large Data Bases*, 327-338, 1992.
9. Hall, F.L. "Traffic stream characteristics," *Traffic Flow Theory, U.S. Federal Highway Administration*, 1996.
10. Hätönen, K., M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, "Knowledge Discovery from Telecommunication Network Alarm Databases," *Proceedings of International Conference on Data Engineering*, 115-112, 1996.
11. Mannila, H., H. Toivonen, and A.I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery,* 1, 3, 259-2, 1997.
12. Ng, A. and A.W.C. Fu, "Mining Frequent Episodes for Relating Financial Events and Stock Trends," *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 27-39, 2003.
13. Wang, F. and P. Liu, "Temporal Management of RFID Data," *Proceedings of International Conference on Very Large Data Bases*, 1128-1139, 2006.
14. Wu, E., Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 407-418, 2006.