History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources

Zheng Li
Oracle Corporation
zheng.zl.li@oracle.com

Tingjian Ge
University of Massachusetts, Lowell
ge@cs.uml.edu

ABSTRACT

Complex event processing (CEP) has proven to be a highly relevant topic in practice. As it is sensitive to both errors in the stream and uncertainty in the pattern, approximate complex event processing (ACEP) is an important direction but has not been adequately studied before. ACEP is costly, and is often performed under insufficient computing resources. We propose an algorithm that learns from the past behavior of ACEP runs, and makes decisions on what to process first in an online manner, so as to maximize the number of full matches found. In addition, we devise effective optimization techniques. Finally, we propose a mechanism that uses reinforcement learning to dynamically update the history structure without incurring much overhead. Put together, these techniques drastically improve the fraction of full matches found in resource constrained environments.

1. INTRODUCTION

Complex event processing has proven to be useful in practice. Complex event matching is sensitive to both errors in data streams and errors or uncertainty in patterns. A complex event pattern p is usually represented as a regular expression extended with a window constraint, allowing interleaving of irrelevant events [5, 9]. Thus, when there is one critical basic event in the stream sequence that is supposed to match a basic event in p but differs due to noise or missing events, then the match will fail altogether. Furthermore, a user who issues the complex event query may not always be exactly sure about the pattern p. She roughly knows what she is looking for, and tries to specify a pattern p. However, if a piece of the stream matches p approximately, it might be an interesting match. Let us look at an example.

EXAMPLE 1. *Twitter is one of the most popular social networks. We can define basic events (i.e., letters), as well as complex event patterns to find complex events of interest in a timely fashion from the Twitter stream. For instance, suppose we want to detect "buy" or "sell" points for the company Apple's stock. The idea is that if there are discussions of bullish stock chart patterns, followed by some good news of the company, we identify it as a buy point (as the stock will likely go up). We can use the following pattern:*

$$p = [Sa(b|d|w|r)E]^{2+}[Sa(g|e|u)E]^{2+} \quad (1)$$

where S is the start of a tweet (identified by @name in the text), E is the end of a tweet (identified by a timestamp), a is the appearance of stock symbol $AAPL (Apple, Inc.), b is "bullish", d is "double bottom", w is "falling wedge", r is "rounding bottom", g is "decent/solid guidance", e is "good earnings", and u is "upgrade". Note that d, w, and r are typical bullish stock chart patterns (more can be included) signifying a potential upward movement of the company's stock price in the future, while g, e, and u are typical good news of a company that ensures the upward movement (these terms are usually posted by experienced traders or trading professionals). The notation "2+" means the pattern in the brackets occurs two or more times. Thus, p finds two or more tweets that indicate a potential upward trend followed by two or more tweets that corroborate it with good news evidence. The whole pattern indicates a "buy" point for Apple.

In Example 1, similarly we can define a complex event for a "sell" point and do this for many companies. Here, the basic events/letters are derived from matching keywords, and the timestamp of a letter is the timestamp of the corresponding tweet. There is clearly uncertainty in the letter sequence since a trader may not use a term that we expect. We are not completely sure about the pattern p; a little deviation from it is probably a good discovery too. We study approximate complex event processing (ACEP). Our semantics is a relaxation of the commonly-used exact version by allowing at most k event errors in the match instance. For instance, in Example 1, we may have $k = 1$ which allows at most one mismatch of the required letters.

Complex event processing can be very resource consuming, as every intermediate matching state has to be kept in the system until either it results in a full match or the match window expires. This process is also nondeterministic, for we want to get all match instances—the actions of moving to the next state (due to a letter match) and staying in the current state are both valid. ACEP makes it even more nondeterministic, since we now also have the choice to skip a letter and go to the next state, adding one to the error. The performance issue is a showstopper when the processing speed cannot keep up with the stream rate.

The aforementioned problem is even more serious if we consider a novel application where more computing is pushed down to endpoint small devices such as smartphones, in order to reduce the amount of communicated data to the server. In particular, an application may want to perform

Proceedings of the VLDB Endowment, Vol. 10, No. 4

such ACEP matching in situ—on smartphones or other devices with limited computing power. Only matched instances are communicated to the remote server. Hence, it is imperative to study *resource-constrained* ACEP (RC-ACEP).

RC-ACEP is a best-effort attempt to discover as many occurrences of the searched patterns as possible, given that the stream rate is higher than can be handled by available computing resources. For RC-ACEP, we propose a novel learning method using a data structure $\mathcal{H}$ we build for the recent match history. We look up $\mathcal{H}$ to determine which intermediate partial matching states are more likely to eventually end up with a full match. We devise an online algorithm and prove its competitive ratio. Furthermore, we propose an optimization technique called proxy match, and devise a novel sketch technique to further reduce the memory consumption. Finally, we use a reinforcement learning technique to dynamically update $\mathcal{H}$ with little overhead. Our contributions are summarized below:

- We propose the ACEP problem and devise an algorithm for it (Sections 2 and 3.1).
- We study the RC-ACEP problem, and propose a novel algorithm that learns from history. (Section 3.2).
- We devise an optimization called proxy match, and a history sketch with its analysis (Sections 3.3 and 4).
- We propose dynamic update of the history structure based on reinforcement learning (Section 5).
- We perform a systematic experimental evaluation using two real world datasets (Section 6).

2. PROBLEM FORMULATION

We are given a *sequence* $s = s[1], s[2], ...$ where each *character* $s[i]$ is in Σ, the *alphabet*. We use the terms *sequence* and *stream* interchangeably, and use *character*, *letter*, and *event* interchangeably. Each $s[i]$ has a *tag* $t[i]$. If $t[i]$ is the timestamp when $s[i]$ is generated, we get *time-window* semantics. If $t[i]$ is a counter (i.e., $t[i] = i$), we get *count-window* semantics. The sequence s has increasing tags. Without loss of generality, we assume time windows. A *subsequence* is a sequence that can be derived from another sequence by deleting some letters without changing the order of the remaining letters. For example, the sequence "bde" is a subsequence of "abcdef".

As common in computer science, a *regular expression* is a pattern descriptive language whose recursive definition has a one-to-one correspondence with the construction of a Thompson's automaton [6] in Figure 1. Similarly, we define a *serialization* of a regular expression p recursively as follows: in the base case, the serialization of a letter a is just the letter a itself, and the serialization of ε is an empty string; the serialization of $\alpha \cdot \beta$ is the serialization of α concatenated with the serialization of β; the serialization of $\alpha|\beta$ is either the serialization of α or the serialization of β, and exactly one of them; the serialization of α^* is 0 or more serializations of α concatenated one after another. For example, for pattern $p = b(c|d)e^*$, one serialization of p is $bdee$, and another one is bc, among many others. We start with defining the notion of approximate complex event processing.

DEFINITION 1. ***(ACEP)*** *Given a sequence s, a complex event pattern p (regular expression), a window size w, and a threshold k, the approximate complex event processing (ACEP) problem is to find each match of p as a subsequence σ (called a critical subsequence) in s within a time window of size w with at most k errors, i.e., σ would be a serialization of p if no more than k letters were added into σ.*

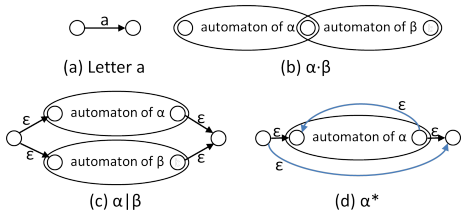


Figure 1: Thompson's automaton. (a) A letter has a start and a final state. (b) Concatenation: α's final state merges with β's start state. (c) Or, $\alpha|\beta$: create new start/final states. (d) Kleene star, α^*: create new start/final states; note the new forward and back edges.

Two remarks are in place. First, the basic syntax of a regular expression only consists of the four components in Figure 1. For brevity, in this paper, we also use commonly used syntactic sugar such as x^{2+} in Equation (1) of Example 1, which is just the shorthand for $xx(x)^*$. Second, Definition 1 exactly corresponds to a *skip till any match* for the *event selection strategy* as categorized in [5]. This model finds the most (non-deterministic) matches. We discuss below (after Example 2) the rationale and support of other models.

EXAMPLE 2. *Suppose $p = a(b|c)db$ with a window $w = 5$ and an error threshold $k = 2$. Let the sequence be $s = a_1e_2e_3b_4c_5e_6a_7e_8e_9c_{10}d_{11}a_{12}e_{13}b_{14}d_{15}$, where a subscript denotes the timestamp of the letter. Then s contains a match with the critical subsequence a_1b_4 (as adding two letters "db" to it will make it a serialization of p). Similarly, s contains another match with error 1 with the critical subsequence $c_{10}d_{11}b_{14}$, as prepending letter "a" would make it a serialization of p.*

ACEP Language Support and Connections with Previous Systems. Note that, in this work, it is not our goal to design a specific CEP language interface; we assume a higher level language similar to SASE [18]. But rather, Definition 1 only succinctly extracts the key language features relevant to the *algorithmic* framework. That is, the ACEP problem in Definition 1 is phrased at an intermediate-representation level below a front-end language like SASE. In a complete CEP system [18, 9], there are concepts of event types and event attributes. However, most of these systems are automaton-based (except ZStream [14], which is query-tree based); thus, an algorithm designed for Definition 1 can be readily integrated into a specific system. Our implementation easily adds the support of predicates: e.g., equivalence tests and parameterized predicates [18], as discussed in Section 3.1. As a result, we currently support CEP language features as in SASE [18] plus *union* (whose implementation is not discussed in [18]) but minus *negation*.

For *event selection strategy*, Definition 1 falls into "skip till any match" as categorized in [5]. However, other strategies described in [5] are easier to implement on top of *skip till any match*. For example, *skip till next match* is similar, but it is a deterministic approach following one fixed match trajectory, while *partition contiguity* requires partitioning the stream into sub-streams and a match node must match the next event in the sub-stream (but must not skip it). Hence, as pointed out earlier, Definition 1 serves as a central algorithmic framework for automaton-based matching, where many other language features can be added on top.

We note that there are richer CEP language features such as negation (which can be added on top of ours), those in SASE+ [5] regarding more complex predicates and HAVING clauses, and the hierarchical structures in XSeq [15]. However, as pointed out earlier, the algorithmic frameworks in the previous work are almost all automaton-based (except ZStream [14] which is query-tree based). Even XSeq is based on the *Visibly Pushdown Automata*, a generalization of finite state automata [15]. Thus, the main contributions of our paper, including using a history structure over the match nodes at automaton states to perform resource-constrained matching, can be easily adopted for a particular instance of CEP system and language. ZSream supports *conjunction* [14], which can be added on top of an automaton-based algorithmic framework too, as done in [13]. An interesting point raised in ZStream [14] is that it is beneficial to have a *flexible* (rather than *fixed*) evaluation order. This indeed can be accomplished in an automaton-based framework too, as done in the *automaton sketch* optimization in [13]. Potentially, our history-based resource-constrained matching can be adopted for a tree-based framework like ZStream as well, where the history would contain tree nodes; we leave this as future work. An extensive survey of CEP language features is beyond the scope of this paper, and we refer the reader to an excellent survey paper [9].

Finally, regarding error metric, *edit distance* [16] is often used as the distance metric between two sequences, in which a unit cost is inflicted for each *insertion*, *deletion*, or *update* of a letter. However, in our case (a match between a pattern and a stream window), for the *skip till any/next match*, the above three operations can be merged to one: an *insertion* to the stream. This is because update of a letter to the stream is the same as insertion, and deletion is free (due to the skips). For other models that require contiguity (i.e., *substring* rather than *subsequence* match), edit distance can be used. A potential generalization of our error model is to give *weights* to each inserted letter that would form a serialization of the pattern, as missing events may have different levels of importance. This is particularly the case if we add *negations* to the patterns, because the presence of a negative event in the stream may get far more penalty than missing a positive event. We now define RC-ACEP.

DEFINITION 2. ***(RC-ACEP)*** *The resource constrained approximate complex event processing (RC-ACEP) problem is the ACEP problem under actual computing resources, with the requirement of giving answers online. Specifically, there is a time constraint determined by the dynamic data stream rate. The online processing must keep up with the stream rate, and return as many matches as possible.*

We focus on the time constraints (i.e., insufficient computing time), although in one variation of our solution we also significantly reduce memory consumption by building a sketch of our data structure. We first present an algorithm for ACEP ignoring resource limits (i.e., the online requirement). Then we concentrate on the RC-ACEP problem.

3. ACEP AND RC-ACEP ALGORITHMS

3.1 ACEP Algorithm

We first devise ACEPMATCH that does the matching without considering the online requirement. We use a Thompson's automaton [6], whose recursive construction is illustrated in Figure 1. A state can either be an L-state (with only one incoming letter edge) or an ε-state (with one or more incoming ε edges). There is a correspondence between any regular expression p and such an automaton; thus ACEP uses such an automaton as the input pattern. A central concept of ACEP is *match trajectory*, which consists of a number of *match nodes*, and indicates a specific critical subsequence in s that causes the match of p. A match node is a tuple:

$$(state, loc, err, start\_win, prev\_state, prev\_loc, prev\_err)$$

where *state* is the current automaton state, *loc* is the location (index) of the letter in the stream s that matches this L-state (if any), *err* is the error (number of mismatches), *start_win* is the start time of this window, *prev_state* is the previous L-state that is matched to the letter at *prev_loc* in the stream with an error *prev_err*. The *prev_** fields are used to chain the match nodes into a match trajectory. The matching algorithm basically keeps tracks of linked match nodes located at each state of the automaton. When one of them is at a final state, the whole match trajectory can be retrieved by following the links backwards. Match nodes expire when they are out of the current window.

Algorithm 1: ACEPMATCH$(s, \mathcal{A}, k)$

Input: s: incoming stream, $\mathcal{A}$: automaton for pattern p, k: error threshold
Output: match trajectories

```
1  seed ← (start{A}, null, 0, null, null, null, 0)
2  propagate(seed, A, k) //Pre-compute this only once
3  for each incoming letter s[i] in s do
4      for each L-state σ of A that matches s[i] do
5          for each match node μ at a state in pre(σ) do
6              if μ.start_win = null then
7                  start_win ← t[i]
8              else
9                  start_win ← μ.start_win
10             if μ.loc = null then
11                 prev_* ← μ.prev_*
12             else
13                 prev_* ← μ.*
14             err ← μ.err
15             create λ ← (σ, i, err, start_win, prev_*) at σ
16             propagate(λ, A, k)
17             if any match node at final state then
18                 report it
1  Procedure propagate(μ, A, k)
2      for each state σ ∈ post(μ.state) in A do
3          if σ is L-state then
4              err ← μ.err + 1
5          else
6              err ← μ.err
7          if err > k then
8              continue;
9          if μ.loc = null then
10             prev_* ← μ.prev_*
11         else
12             prev_* ← μ.*
13         create λ ← (σ, null, err, μ.start_win, prev_*) at σ
14         propagate(λ, A, k)
```

In line 1 of ACEPMATCH, we create a single "seed" match node at the start state of the automaton, $start\{\mathcal{A}\}$, with error 0 and other fields *null* (the format of a match node is shown earlier). All other match nodes later created during the algorithm are descendants of this single seed. Note that

the start state is an ε-state, which is why it does not consume any input letter for the seed to be on. For an L-state with incoming letter l, a match node can be on it only after consuming a letter l from the stream, or after skipping the letter l (with an additional error 1 incurred).

Line 2 calls the *propagate* procedure to propagate and derive new match nodes from the seed. This is to skip up to k L-states (each incurring error 1) and generate new match nodes. Line 2 of the *propagate* iterates through each *post* state (i.e., a state that immediately follows the current state) and tries to generate a new match node based on the original one. An error 1 is incurred if the new state is an L-state. In lines 9-12 of *propagate*, it first checks if μ's stream location is *null*. If so, it is not an L-state match there, and hence the $prev\_*$ fields (i.e., $prev\_state$, $prev\_loc$, and $prev\_err$) of the new match node (created in line 13) should be taken from μ's $prev\_*$ fields (line 10). Otherwise there is an L-state match at μ, and we set the new $prev\_*$ fields to μ's corresponding fields $(state, loc, err)$ (line 12). Finally, in line 14 of *propagate*, the procedure is recursively called over the new match node until it cannot be propagated further (either when error is over k or at a final state).

Lines 3-4 of ACEPMatch iterate through each letter $s[i]$ and each L-state σ that matches the letter. This match will trigger the creation of a new match node at σ (line 15). Such a new match node is based on an existing match node μ at a previous state in line 5, where $pre(\sigma)$ denotes the set of states that immediately precede state σ. Note that if a match node μ in line 5 has expired, it is simply removed. Lines 10-13 are to set the $prev\_*$ fields of this new match node (same as lines 9-12 of *propagate*).

Example 3. *As in Example 2, let $p = a(b|c)db$, $w = 5$, $k = 2$, and the automaton is shown in the top plot of Figure 2, where S1 is the start state and S6 is the final state. The bottom plot illustrates* ACEPMatch*, where a match node is simplified as a two-value tuple only (state and error). Stream events with timestamps as subscript are shown vertically ($c_1 d_2 a_3 e_4 b_5$). $(S1, 0)$ is the seed node before any events arrive, with error 0. Line 2 of* ACEPMatch *propagates the seed to the three match nodes to the right of it in the same row. When c arrives at time 1, the matching L-state S4 is examined (line 4) whose pre set only has S2. Thus, a new match node $(S4, 1)$ is created based on $(S2, 1)$. A solid arrow indicates such a "match and advance" action, corresponding to (the reverse of) the $prev\_*$ chain pointers, while a dashed arrow indicates skipping a letter during propagate. The red arrows together form a match trajectory with error 1 at time 5. The blue arrows form another match trajectory with error 2. This example shows that there can be many very similar match trajectories. Our subsequent techniques (Sections 3.2 and 3.3) will address this issue.*

Note that we easily support predicates for the higher level language SASE [18], which is not shown in ACEPMatch for clarity. For a predicate that involves a number of event variables $X_1, ..., X_c$, whenever an event variable is matched, we record the attribute value in the match node. When the last event among the c events is matched, we check the predicate and do not form a new match node if the predicate is false. For instance, suppose we have a predicate $D.att1 > A.att2$ in the pattern of Figure 2, where D and A are the event types/variables of d and a, respectively. Then we record $A.att2$ in a match node and evaluate the predicate

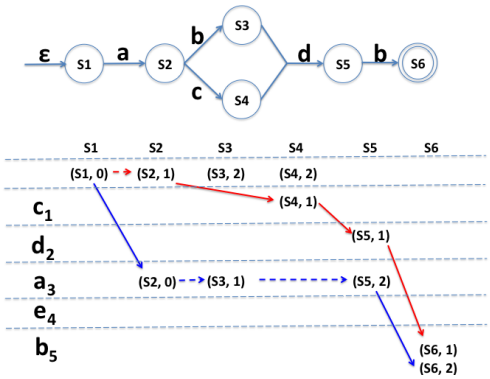


Figure 2: An automaton and the ACEP matching algorithm. We show two of the match trajectories, one in red and the other in blue. A solid edge indicates a letter match, and a dashed edge is a propagation with error.

when a match is triggered in state $S5$. This is analogous to the "predicate pushdown" optimization in [18].

Connection with SASE [18]. Both ACEPMatch and SASE matching use an NFA. We next show that there is an interesting connection between the two: *When the error threshold k is 0,* ACEPMatch *is the same algorithm as the SASE along with all the optimizations presented in [18], plus the union support, and minus the negation support.* We observe that there is a one-to-one correspondence between the supported features/algorithm components of SASE (with the optimizations in [18]) and ACEPMatch when $k = 0$. The Sequence Scan and Construction (SSC) using Active Instance Stack (AIS) (presented in Section 4.1 and Figure 3 of [18]) is algorithmically equivalent to our *match nodes* and *match trajectories*. Specifically, an element in the AIS associated with an automaton stack corresponds to our match node, while an arrow/pointer in Figure 3 of [18] corresponds to our $prev\_*$ pointers used to trace back the match trajectories. When $k = 0$, the *propagate* procedure in ACEPMatch will never propagate a match node to a following L-state as there is no error budget and *propagate* does not consume any event—thus SASE and ACEPMatch progress among the states in the same manner and complexity.

Earlier we have discussed how our algorithm handles predicates. Our framework does the same "pushing an equivalence test down" optimization (Partitioned Active Instance Stacks in Section 4.2.1 of [18]) by running an instance of ACEPMatch for each partitioned sub-stream. The *cross-attribute transition filtering* in Section 4.2.2 [18] corresponds to our discussion above where a predicate is evaluated when the last event type in the pattern is matched. Moreover, ACEPMatch does the "pushing windows down" optimization in Section 4.3 of [18], as it keeps track of the start of a window in each match node, which is removed when the window expires. In all, we have the algorithmic equivalence as stated earlier. Of course, the design of ACEPMatch is for the main goal of approximate match when $k \neq 0$.

3.2 History and RC-ACEP

As discussed earlier, ACEP can easily exceed resource limits. We now present the RC-ACEP algorithm that does the best-effort most-profitable online processing. The basic idea is that we can somehow compactly represent a *history* of past run into a data structure $\mathcal{H}$. At runtime, based on $\mathcal{H}$,

we discard some match nodes that are unlikely to result in a final full match. However, this is not a straightforward problem where we can apply an off-the-shelf machine learning technique. We illustrate this point below.

At any moment, there are many active match nodes that are ready to be expanded. A match node may wait at an automaton state, or it can skip to a subsequent state but incur an error, or it can simply be discarded. When there is a letter match, there may also be a choice whether to advance into a union branch or not. Hence, there are many choices/decisions to make. Letting each match node *independently* decide what choice to make would not work well as a whole. This is because they are heavily correlated—a sequence of future events in the stream may make multiple match nodes to complete a full match together (in the same time window) or mutually exclusively.

One example is the match nodes at the same or nearby automaton states. There are slight variations of essentially the same matches. For instance, two matches may differ only by one letter or by one error. The correlation between two match trajectories is caused by the same set of close events that occur in the stream. The key insight is that, if two match trajectories are within the same window (or close by), we only need to keep one match node and discover one of the trajectories. It is more important to detect *which time window* has the match; this is often sufficient already, especially given the minor differences among the trajectories. Once we locate the window, if we really needed to enumerate all trajectories, we could simply do so for that stream window only—which incurs significantly less cost than maintaining all match trajectories throughout their lifetimes. Thus, from $\mathcal{H}$, we learn if two match nodes tend to produce match trajectories that are in the same window or close together, so that we only keep one of them.

Definition 3. ***(Match Point, Match Bundle)*** *The time (tag) of the last event in the stream s that matches an L-state in p and completes a match trajectory τ (i.e., reported in line 18 of* ACEPMatch*) is called a match point of τ, denoted as $m(\tau)$. A match bundle is a set T of match trajectories such that: (1) (Proximity) if $|T| > 1$, then $\forall \tau \in T, \exists \tau' \in T, \tau' \neq \tau, |m(\tau) - m(\tau')| \leq w$; (2) (Maximality) T is maximal in that $\forall \tau \in T, \forall \tau', |m(\tau) - m(\tau')| \leq w \Rightarrow \tau' \in T$.*

The above definition indicates that a match bundle contains a contiguous region of match points that spans an *arbitrary* time interval. The proximity property says that each match point has *at least* another one that is close (unless the whole bundle has only one match point), while the maximality property says that the bundle cannot be further expanded on either side in time. Thus, any two match bundles in a stream are non-overlapping; otherwise they would be merged into one.

Example 4. *In Figure 2 (Example 3), the upper match trajectory (red) has a match point of 5, which is the timestamp of the last event b_5 that triggers its last L-state match. Similarly, the lower match trajectory (blue) also has a match point of 5. These two match trajectories are in the same match bundle, as the distance between their match points are clearly within w.*

All match trajectories in a stream are partitioned into distinct match bundles. Intuitively, a match bundle indicates an area in the stream that contains matches. Once we locate a match point t anywhere in a match bundle, if all match trajectories were to be retrieved, we could extend t on both sides of the stream (forward and backward) by w at a time, and do a match in the local area of t until an extension resulted in no more matches. By the definition of a match bundle, it is easy to see that two closely correlated match nodes (that tend to either both succeed or both fail to lead to a full match—due to the chance in event sequence) will also tend to lead to the same match bundles in the history $\mathcal{H}$. Hence, match bundles are a crucial part of a history.

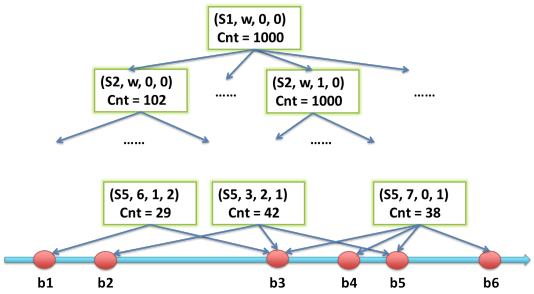


Figure 3: Illustrating the history data structure $\mathcal{H}$. Each green box is a match vector type of vertex, and each red ball is a match bundle type of vertex, occurring during some time interval in the stream.

Definition 4. ***(History)*** *A history $\mathcal{H}$ for a time interval $[t_1, t_2]$ is a directed graph with vertex labels. $\mathcal{H}$ has two types of vertices: (1) a match vector $(state, remaining\_w, err, time\_since\_last)$, where state is the current state in the automaton, $remaining\_w$ is the remaining window size, err is the currently incurred match error, and $time\_since\_last$ is the time since the last event match within a match trajectory, and (2) a match bundle. In addition, a match vector vertex is also labeled with an integer count of visits, while a match bundle vertex is labeled with its ID (unique in the stream). An edge between two match vectors indicates a visit sequence from one to the other, while an edge from a match vector to a match bundle indicates that the match vector directly leads to the match bundle.*

Example 5. *Figure 3 illustrates the history data structure $\mathcal{H}$ learned from a period of run of the pattern in Example 2. The green rectangles are type (1) vertices in Definition 4—match vectors, while the red balls at the bottom level are type (2) vertices—match bundles in time order of the stream. A match vector is essentially a state of a particular match node ($remaining\_w$ and $time\_since\_last$ are discretized into integers for various interval sizes in a window), and the count is the number of times reaching that vector state during the period of run. Many vertices are omitted from the figure. Note that some match vectors may not have outgoing edges, corresponding to the expiration of the match node without full matches. Moreover, one match bundle may have incoming edges from multiple match vectors, indicating that those match vectors are correlated and had match instances in the same match bundle in the stream.*

A common operation we will perform on $\mathcal{H}$ is to look up the list of match bundles reachable from a given match vector. Thus, for fast lookup, we associate with each match vector vertex a bitmap called a *match map*. A match map is a bitmap with one bit for each match bundle in $\mathcal{H}$. A bit is set to 1 if the corresponding match bundle is reachable from that match vector. For example, in Figure 3, every match vector node has a bitmap (match map) of 6 bits. The match map at the match vector $(S5, 3, 2, 1)$ has a match map

011010, as it has b_2, b_3, and b_5 as descendants. The $\mathcal{H}$ here is small for clarity. In a real structure, typically each match map has hundreds or thousands of bits.

$\mathcal{H}$ can be easily built while we run the ACEP matching algorithm on the data stream. As each match vector state is encountered, we increment its count in $\mathcal{H}$. Whenever there is a full match, we trace back the match trajectory and set bit 1 for the corresponding match bundle in each match map of the match vector node on the trajectory.

Given a history $\mathcal{H}$, we solve the RC-ACEP problem. The basic idea is as follows. We must pick more "promising" match nodes and process them first. This is done in an online manner such that, when the next stream event arrives (before we finish processing all match nodes), we stop and process this new stream event. Thus, the question becomes: given a set of currently active match vectors, which ones should we pick in an online manner so that the number of full matches can be maximized? $\mathcal{H}$ shows the set of match bundles each active match vector led to in the past, through the match map at that match vector.

Intuitively, this problem can be reduced to a variant of the *maximum coverage problem* [11]. Recall that the maximum coverage problem is: One is given several *sets* and a number θ; the sets may have some elements in common. One must select at most θ of these sets such that the maximum number of elements are covered. In RC-ACEP, upon each stream event, each *active match vector* maps to a *set*, where each *element* in a set is a match bundle in $\mathcal{H}$. Since we cannot handle all the active match vectors, we want to pick a limited number of them so that they cover as many match bundles in $\mathcal{H}$ as possible.

However, there are two major differences: (1) In the maximum coverage problem, we get to pick a fixed number (θ) of sets to maximize the number of elements covered. In RC-ACEP, due to the highly dynamic nature of stream rates and system resource environment, we do not know in advance how many match vectors we can finish. We must process them in an online manner. (2) The match map statistics at each match vector in $\mathcal{H}$ is biased by its visit count. That is, if vector v_1 is visited less often in history than vector v_2, it tends to have fewer "1" bits (i.e., match bundles) in its match map. But *given* the fact that they are both active at the current moment, we should remove this bias and use the statistics *conditioning* on the fact that are both active.

For instance, in Figure 3, suppose at present two match vectors $v_1(Cnt = 29)$ and $v_2(Cnt = 42)$ are active. Then we normalize v_1's match bundle counts by multiplying them with $42/29 = 1.45$. That is, choosing v_1 covers b_1 and b_3 1.45 times, while choosing v_2 covers b_2, b_3, and b_5 1 time. Choosing both still covers b_3 1.45 times (multiset union). Thus, our solution generalizes *sets* to *multisets* in the maximum coverage problem.

The algorithm is shown in Algorithm 2. We say that a letter is a *matching letter* if it matches at least one L-state in the automaton. Lines 1-2 are the same as ACEPMatch, setting up the seed match nodes. In line 3, we use the keyword "immediately", meaning that whenever a new matching letter arrives, we must stop the block of code below line 3 (even if it is not finished), and process this new letter. Line 4 initializes an array max[.] to all 0's. The size of this array is the same as the number of bits in a match vector. It is used to keep track of the maximum number of times a match bundle has been covered so far.

Lines 5-11 iterate through each match node at the "pre" state of each L-state that matches the incoming letter. These match nodes are candidates. We will later pick and process them one by one until time is out and the next letter arrives. In line 7, we look up $\mathcal{H}$ and get the match node's match vector's information: number of visits and match map. Line 8 is to get the multiplicity of each bundle covered by μ as discussed above. z is a scale constant that can be set to any value (e.g., maximum count, for numerical accuracy). In line 9, $1(b)$ denotes the number of 1's in bitmap b; with multiplicity, *score* is the initial total count of bundle coverage.

Algorithm 2: RC-ACEP$(s, \mathcal{A}, k, \mathcal{H})$

```
   Input: s: incoming stream, A: automaton for pattern p, k:
          error threshold, H: history
   Output: match trajectories
1  seed ← (start{A}, null, 0, null, null, null, 0)
2  propagate(seed, A, k)//Pre-compute this only once
3  for each arriving matching letter s[i] immediately do
4      reset max[.]
5      for each L-state σ of A that matches s[i] do
6          for each match node μ at state pre(σ) do
7              get μ's count c and match map b from H
8              m ← z/c //z is a constant; m is multiplicity
9              score ← 1(b) × m
10             global_version ← 1
11             put (μ, m, b, global_version, score) into priority
                 queue Q with priority score
12     while Q is not empty do
13         (μ, m, b, version, score) ← pop maximum from Q
14         updates ← look up version table using version
15         if updates & b ≠ 0 then
16             score ← Σ_{i∈b} max(m − max[i], 0)
17             put (μ, m, b, global_version, score) back to Q
18         else
19             process μ as in ACEPMatch lines 6-18
20             update max[.], global_version, version table
```

The algorithm uses a simple version table, which is just an array associating a version number (starting from 1) with a bitmap of updates in $max[]$ since that version. This is to efficiently keep track of whether scores are up to date during the greedy selection of match nodes. Lines 10-11 put the initial version of match nodes into a heap (priority queue). Lines 13-15 get the maximum score match node μ from the heap, and look up the positions of updates in $max[.]$ since its version. If they overlap μ's match map b, in line 16, we update μ's *score* to a smaller one, removing the counts of match bundles already covered since the previous version. Otherwise, in line 20, we update $max[.]$ for newly covered bundle counts, increment the $global\_version$, and update the update-maps of the version table.

Overall, lines 4-20 are a novel integration of the online greedy maximum coverage algorithm and an A*-style search [12]. The goal is to pick which match vectors to process in an online manner to maximize the total (normalized) count of match bundles covered by the selected match vectors. The A* search in particular is to efficiently search for a match vector that gives the maximum coverage increment of match bundles. Thus, the *negative value* of the *score* of each active match node is regarded as the f cost function in the standard A* model [12], where *score* is the coverage increment since *any version* (as recorded in Q). It is the sum of two

cost functions (g and h in the A* model), where g corresponds to the coverage increment since *global_version*, and h corresponds to the coverage between the version recorded in $\mathcal{Q}$ and *global_version*. Thus, *score* is always an *upper bound* of the true increment, and the algorithm is correct.

EXAMPLE 6. *Suppose we have 3 candidate match nodes μ_1, μ_2, and μ_3 (lines 5-6), and the result of looking up $\mathcal{H}$ in line 7 is $c_1 = c_2 = c_3 = 168$ and $b_1 = (1000)$, $b_2 = (0110)$, $b_3 = (0111)$ (μ_i have visit count c_i and match map b_i). For clarity of illustration, let $z = 168$ (z can be any constant); so $m = 1$ for all three candidate match nodes. In line 9, initially their scores are 1, 2, and 3 resp., and they enter the priority queue in line 11. Initially in line 4, max[.]'s all four entries are 0. In line 13, the first match node popped out must be μ_3 as it has the highest initial score. Indeed it is the most "promising" one as it led to the most matches in $\mathcal{H}$. Note that the current version table is empty, and hence the updates bitmap of line 14 is empty. Thus line 19 will process μ_3. In line 20, max[.] array is now (0, 1, 1, 1), the global_version becomes 2, and version table associates "version 1" with an update-bitmap (0111) for what is changed in max[.] since version 1. Then the next round line 13 will pop out μ_2 as it has a higher score (2) than μ_1. μ_2's version is 1; looking up the version table finds updates to be (0111) since version 1 (line 14). As this intersects b_2 (line 15), we re-calculate its score to be 0 (no new bits) in line 16, and it is put back into $\mathcal{Q}$. Similarly, the next round line 13 will pop out μ_1 which has score 1. Line 14 "updates" is (0111) which does not intersect b_1, implying that its score is exact. So μ_1 is processed in line 19. Line 20 updates max[.] to be (1, 1, 1, 1), increments global_version, and the version table now has: version 1 with (1111) and version 2 with (1000). The last match node to be processed is μ_2.*

The RC-ACEP algorithm is an online algorithm; we perform a *competitive analysis* with regard to the total count of match bundles covered. The proofs of all the theorems in this paper appear in our technical report [3].

THEOREM 1. *The online algorithm RC-ACEP has a competitive ratio of $1 - \frac{1}{e}$. Furthermore, it is optimal in the domain of polynomial time algorithms, even the offline ones (unless $P = NP$).*

3.3 An Optimization

We now present an optimization technique that stems from the idea that we do not need to keep track of all matching trajectories that are very close. Instead, we only record the most "promising" partial matches at each automaton state. This is done in such a way that if they do not produce a final full match, then no other partial matches can. Moreover, once we find a full match resulted from one of the promising partial matches, if we choose to recover all full matching trajectories close to the one we find, we can efficiently do so. In other words, this optimization does not lose any matches. Then the key point is what constitutes the *promising match nodes* at each automaton state.

DEFINITION 5. ***(Proxy Match)** A proxy match optimization is one in which we maintain fewer match nodes, known as proxies, than the original algorithms. If there is a window in the stream that contains at least one match found by the original algorithm, then proxy match can find that window, and a match with the shortest duration in that window.*

We next show a specific strategy for proxy match, together with its properties. We say that an L-state s_1 precedes another L-state s_2, denoted as $s_1 \prec s_2$, if in any possible match trajectory, s_1 is reached before s_2.

THEOREM 2. *The following strategy is a proxy match: at each L-state of the match automaton, we only keep one match node per error level $i = 0, ..., k$, which has the greatest (i.e., latest) start_win among all match nodes there with error at most i. We write ${w_i}^s$ as the start_win of the match node of error level i at state s. Then (1) for any state s, ${w_0}^s \leq {w_1}^s \leq ... \leq {w_k}^s$; (2) for any error level i, and for any two states $s_1 \prec s_2$, we have ${w_i}^{s_1} \geq {w_i}^{s_2}$.*

The proxy match strategy described in Theorem 2 can be seamlessly integrated into ACEP or RC-ACEP, and stores at most $k+1$ match nodes at each automaton state. In addition, if all match trajectories are needed by an application, the most recent window of events in the stream is always stored in memory—so that it can be used to find all match trajectories once the proxy is found to match.

4. SKETCHING HISTORY

The history data structure $\mathcal{H}$ can be very large; thus we devise an effective sketch for $\mathcal{H}$. Our sketch stems from a count-min sketch [8], but we modify it to incorporate match map information. Recall that a count-min sketch is represented by a two-dimensional array counts with length l and depth d: $count[1,1]$ ...$count[d,l]$. Additionally, it uses d hash functions $h_1...h_d : \{1...n\} \rightarrow \{1...l\}$ chosen uniformly at random from a pairwise independent family. We modify it such that each cell of the two-dimensional array is not just a count $count[i,j]$, but it also includes a bitmap $b[i,j]$ of m bits. The hash functions $h_1...h_d$ are applied over the match vectors $(state, remaining\_w, err, time\_since\_last)$.

Algorithm 3: BUILDSKETCH$(s, \mathcal{A}, k)$

```
   Input: s: incoming stream; A: automaton for pattern p; k:
          error threshold
   Output: history sketch
1  for each new match vector v during ACEPMATCH(s, A, k)
     do
2      for each i ← 1...d do
3          j ← h_i(v)
4          count[i, j] ← count[i, j] + 1
5      if v is at a final state then
6          for each match vector u traced back from v do
7              for each i ← 1...d do
8                  j ← h_i(μ)
9                  set a bit in b[i, j] to be 1 for this match
                     bundle
10 return count[·, ·] and b[·, ·]
```

In lines 2-4 of BUILDSKETCH, we apply the d hash functions on the match vector v. For a hash value j from the i'th hash function h_i (line 3), we increment the counter at row i column j. Initially all counters are 0. When we have a full match (line 5), we trace back all the match vectors in the trajectory, applying the d hash functions, find the d cells in the sketch, and set the bit corresponding to this match bundle to be 1 in all bitmaps of the d cells (lines 8-9). The usage of the sketch is in LOOKUPSKETCH. It basically puts the minimum count into c, and the intersection (bitwise AND) of all d bitmaps into bm, which are returned. Figure

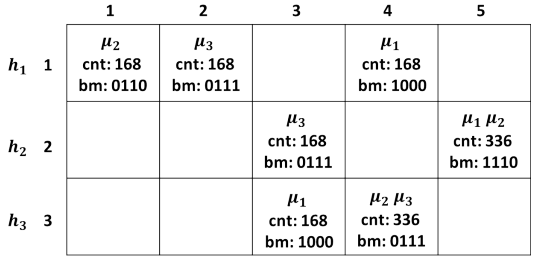


Figure 4: Illustrating the history sketch.

4 illustrates a sketch, where multiple match vectors (e.g., μ_1 and μ_2) may collide in the same cell. The algorithm will return μ_1's visit count as 168 and match map as 1000.

Algorithm 4: LookUpSketch$(v, count[\cdot,\cdot], b[\cdot,\cdot])$

Input: v: match vector; $count[\cdot,\cdot], b[\cdot,\cdot]$: history sketch
Output: visit count and match map of v

```
1 c ← count[1, h_1(v)]
2 bm ← b[1, h_1(v)]
3 for each i ← 2...d do
4 |   j ← h_i(v)
5 |   c ← min(c, count[i, j])
6 |   bm ← bm & b[i, j]
7 return c and bm
```

Let us analyze the history sketch. It is clear that both the counts and the match map may only have positive, but not negative, errors (i.e., increased counts and more bits set). This bias is alleviated by the fact that RC-ACEP only relatively compares the counts and match maps of two match vectors to determine which one is processed first.

Theorem 3. *Consider a match vector v. If* LookUpSketch *gives a false positive on its i'th bit of the match map, then the returned count value also has a positive error. The converse may not be true.*

Theorem 3 indicates that an error of any bit in the match map implies an error in the count. Since we typically have many bits in the match map, it is more significant to optimize the parameter choice of the sketch by minimizing the error probability on each bit. Since each cell takes the same amount of space, we study the problem that given a space budget $d \times l = c$, how to choose d and l in the history sketch so as to minimize this error probability.

Theorem 4. *Given a space constraint $d \times l = c$, the parameters that minimize the probability of a bit i being a false positive in the match map of a match vector is $l = \frac{\alpha}{ln2}$ and $d = \frac{c}{a} ln2$, where α is the number of match vectors on the path to match bundle i in $\mathcal{H}$.*

Different match bundles may have different α values. We can use their average for choosing d and l.

5. ADAPTATION TO DATA STREAMS

While we use $\mathcal{H}$ to make decisions for RC-ACEP, for each match-vector vertex in $\mathcal{H}$ (likewise for each array-cell of a history sketch), we accumulate a second copy of match map and visit count. Each bit in the secondary match maps corresponds to a new match bundle found. When the new match bundles fill up the secondary match map (which has a fixed size) completely, we switch to this new/secondary set of match maps and visit counts for RC-ACEP, and this iterative process continues. However, when we use $\mathcal{H}$ to pick match nodes/vectors, we always tend to follow the most promising paths according to the existing $\mathcal{H}$, which may become sub-optimal as time goes and stream trend changes—some match nodes may never get chosen due to time constraints. There is no mechanism to systematically explore the vector space outside, which may become better. We propose a novel method to update $\mathcal{H}$. We say that each match vector is an *agent*. In RC-ACEP, many agents collectively make decisions on who proceeds. The basic idea is that while running the collective decision making, in the background, we run a few *lightweight* agents who do not branch out to multiple match trajectories, but each of them decides a single trajectory towards the final state. They are powered by reinforcement learning (RL) [17]; we call them RL agents.

We need to run RL for individual agents rather than all agents together to avoid too large a state space and action space. The RL agents are rewarded (i.e., reinforced) for discovering new promising match trajectories due to stream trend shift. The trajectory of an RL agent is also written in $\mathcal{H}$. Hence, once *new* matching trajectories leading to full matches are discovered by RL agents, they are further picked up, adopted, and corroborated by collective decision making agents (Section 3). In this way, $\mathcal{H}$ is updated. Q-learning is a type of RL; we refer the reader to [17, 3] for details.

A key challenge is how to map the behavior of an agent to the *states* and *actions* in Q-learning. For an L-state σ, denote $post(\sigma)$ as the set of L-states that immediately follow σ in the automaton without considering back edges. For σ, its Q-learning states are of the form $(\sigma, remaining\_w, err, time\_since\_last, next\_letter)$, where $remaining\_w$, err, and $time\_since\_last$ have the same meaning as before, and $next\_letter$ can be any letter in $post(\sigma)$ or "others". Let $|post(\sigma)| = p$. At L-state σ, there are $2p+1$ actions, corresponding to "advance to post-state 1", ..., "advance to post-state p", "skip to post-state 1", ..., "skip to post-state p", and "stay put", respectively. Algorithm 5 sets up the initial Q function.

Algorithm 5: UpdateHistorySetup$(\mathcal{A}, k)$

Input: $\mathcal{A}$: automaton for pattern p; k: error threshold
Output: $Q[\cdot,\cdot]$

```
1  for each state σ of A in reverse topological sort order do
2  |   post(σ) ← ∅
3  |   dist(σ) ← ∞
4  |   for each succeeding state φ of σ do
5  |   |   if φ is an L-state then
6  |   |   |   post(σ) ← post(σ) ∪ φ
7  |   |   |   if dist(σ) > dist(φ) + 1 then
8  |   |   |   |   dist(σ) ← dist(φ) + 1
9  |   |   else
10 |   |   |   post(σ) ← post(σ) ∪ post(φ)
11 |   |   |   if dist(σ) > dist(φ) then
12 |   |   |   |   dist(σ) ← dist(φ)
13 |   if σ is an ε-state then
14 |   |   continue
15 |   for each state φ ∈ post(σ) do
16 |   |   for each err ← 0 to k do
17 |   |   |   for r_win ← 1 to i_w do
18 |   |   |   |   qs ← (σ, err, r_win, letter(φ))
19 |   |   |   |   for each φ' ∈ post(σ) \ {φ} do
20 |   |   |   |   |   Q[qs, "advance to letter(φ')"] ← −∞
21 |   |   |   |   Q[qs, "skip to letter(φ)"] ← −∞
```

Lines 1-12 compute the set of following L-states ($post$) for each automaton state, and its shortest distance (in number

of edges) to a final state (the $dist(\cdot)$ function). Without considering back edges of the automaton, its graph is a DAG. We later use the $dist(\cdot)$ function to assign rewards. Lines 13-21 initialize the Q-function table to all 0's except some illegal actions. For example, advancing to a post-state that is different from the *next_letter* specified in the Q-learning state is illegal, for which we give Q value $-\infty$.

We now look at UPDATEHISTORY (Algorithm 6). When a new match node is created from a seed, we mark it as a Q-learning agent (line 3). In line 5, qs is the Q-learning state based on information of μ. The $f(Q[qs,a], n[qs,a])$ function in line 6 is called the *exploration function* [17], which determines how greed (preference for high values of Q) is traded off against curiosity (preference for low values of n—actions that have not been tried often). f should be increasing in Q and decreasing in n. Similar to [17], we use a simple one: $f(Q[qs,a], n[qs,a]) = \begin{cases} \infty & \text{if } n[qs,a] < 5 \\ Q[qs,a] & \text{otherwise} \end{cases}$. This has the effect of trying each action-state pair at least five times.

Algorithm 6: UPDATEHISTORY$(s, \mathcal{A}, k)$

Input: s: incoming stream; $\mathcal{A}$: automaton for pattern p; k: error threshold
Output: history sketch

```
1  for each matching letter s[i] do
2      if s[i] moves a seed match node to a new state then
3          mark new match node as Q-learning agent
4      for each match node μ marked as Q-learning agent do
5          qs ← (σ, err, r_win, s[i]) based on μ
6          action ← argmax_a f(Q[qs,a], n[qs,a])
7          σ ← next(σ, action)
8          if σ changed then
9              r ← R/2^dist(σ)
10             move μ to state σ of A
11         if qs' ≠ null then
12             n[qs', a'] ← n[qs', a'] + 1
13             Q[qs', a'] ← Q[qs', a'] + α(n[qs', a'])(r' +
                 γ max_a Q[qs, a] − Q[qs', a'])
14         (qs', a', r') ← (qs, action, r)
15         update history H or its sketch based on σ
16         report a match if σ is a final state
```

Line 7 determines the next state based on the chosen action, while line 9 assigns the reward r if moving to a new state. For a constant R, r decrease exponentially with the distance to final states. Lines 11-13 give the adjustment of $Q[qs', a']$ based on the current estimate of the next state's optimal action's Q value. The $\alpha(n[qs', a'])$ is the *learning rate* function, which decreases as the number of times a state has been visited ($n[qs', a']$) increases. Like [17], we use $\alpha(n[\sigma, a]) = 60/(59 + n[\sigma, a])$. It iteratively updates the Q table, which gives the optimal action policy for the RL agent, adaptive to dynamic changes of streams.

6. EXPERIMENTS

6.1 Datasets and Setup

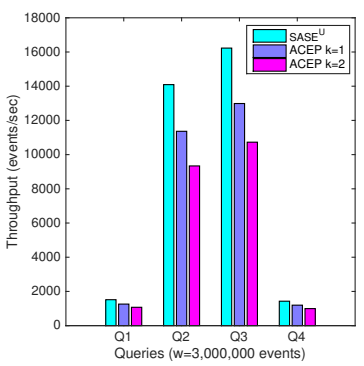
We use two real world datasets: **(1) Twitter Data.** We use the Twitter Stream API [1] and twenty most common words [2] as keywords (which results in a high data rate) to download two weeks of tweets, resulting in a data size of 24 GB. Each tweet message tuple has a fixed format: user ID, tweet text and timestamp, from which events are extracted. The data rate is on average about 110 tuples per second.

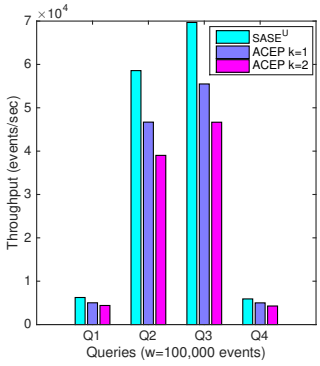
(2) PHEV Data. This is a public test dataset for plug-in hybrid electric vehicle (PHEV) applications in smart grid developed by Akhavan-Hejazi et al. [7, 4]. It contains driving traces for 536 GPS-equipped hybrid electric taxi vehicles for 3 weeks in San Francisco, CA. The dataset is about 800 MB, and the data rate is about 54 events per second. It consists of information such as state-of-charges, charging loads at different identified charging stations, timestamp, etc. There is a revenue opportunity for PHEV fleet owners (such as taxi or rental companies) if unused vehicles give electricity back to the grid which can provide ancillary services, such as load leveling, regulation and reserve [10]. We implement all our algorithms presented in this paper in Java. All experiments are performed on a machine with an Intel Core i7 2.50 GHz processor and an 8GB memory.

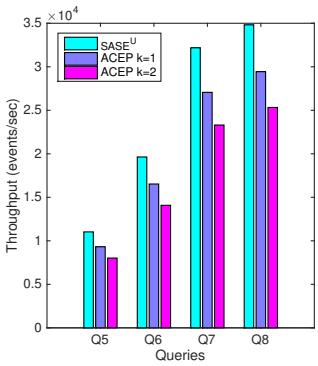
6.2 Experimental Results

We issue the following queries to the Twitter data. **Query** $Q1$ is: $[Sa(b|d|w|r|h|t|c)E]^{2+}[Sa(g|e|u)E]^{2+}$ (similar to Example 1), where most of the letters are described in Example 1, with a few additions: h is "head and shoulder bottom", t is "ascending triangle", and c is "cup with handle", which are all bullish stock chart patterns. As noted in Example 1, these basic events (i.e., letters) are derived from matching keywords with tweets (there are 43 letters in total that we extract from the tweets), and the timestamp of a letter is the timestamp of the corresponding tweet. **Query** $Q2$ is to find when Apple has a bearish stock chart pattern: $Sa(t_1|t_2)ESa(R|H)ESa(B|D)E$, where t_1, t_2, R, H, B, D denote "double top", "triple top", "rising wedge", "head and shoulder top", "bump and run reversal", and "descending triangle", respectively. Similarly, we have **query** $Q3$: $SGgESGeESGuE$ where G is the stock symbol \$GOOGL (Google), and other letters are as before. $Q3$ detects the events of good news of Google. Finally, **query** $Q4$ detects the bullish patterns of one company with the bearish patterns of the other: $[Sa(b|d|w|r|h|t|c)E\ SG(t_1|t_2|R|H|B|D)E]$ | $[SG(b|d|w|r|h|t|c)E\ Sa(t_1|t_2|R|H|B|D)E]$, which may indicate a chance of trading stocks between the two companies.

Query $Q5$ is for the PHEV Data and is to locate a candidate vehicle to charge the grid: $(lp^{5+}(d|c|a))|(p^{5+}n(d|c|a))$, where l is drawing a large amount of power from the station in the past minute (> 2.5), p is in parking status, n is no power drawn from the station in the past minute, and d, c, a are location at downtown, cab depot, and airport, respectively. Intuitively, if a vehicle has drawn a large amount of power, has stayed in parking position for a while, and it is in one of our interested locations, or if it has parked at a charging station for a while and then no power is drawn (as it is in full power), then such a vehicle may be a candidate to charge the grid back. In addition, there is an equivalence test [18] requiring that all matching events have the same vehicle ID. **Query** $Q6$ is $(zL)^{5+}$, also with an equivalence test on vehicle ID, where z is for speed 0 and L is for a low speed in [2, 10] mph, which indicates a situation of traffic jam. **Query** $Q7$ is zLM_1M_2H with an equivalence test on vehicle ID, where M_1, M_2, and H are medium speed in [11, 20] mph, medium speed in [21, 30] mph, and high speed above 30mph, respectively. $Q7$ may signify that a taxi is starting a trip. Finally, **Query** $Q8$ is $lllll$ (recall that l is drawing a large amount of power from a station), with an equivalent test on *location* and each l event has a predicate that its vehicle ID is different from previous l matches. Thus, $Q8$ indicates







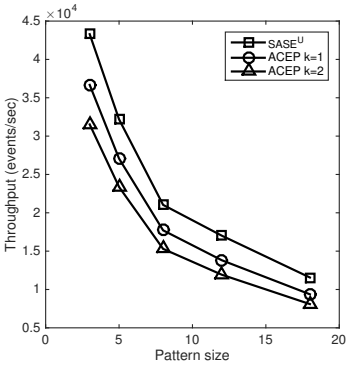


Fig 5 Throughputs (Twitter) **Fig 6 Throughputs (Twitter)** **Fig 7 Throughputs (PHEV)** **Fig 8 Varying pattern size**

that a particular station has been busy—at least 5 vehicles have drawn large amounts of power there.

In the first set of experiments, we examine the performance of ACEP. We first use the Twitter Data and measure the system throughput in number of events the algorithm can process per second. We have discussed in Section 3.1 that, when the error threshold k is 0, ACEPMatch is algorithmically equivalent to SASE with optimizations in [18], plus *union* but minus *negation*. As many queries in our experiments do require *union*, we denote as $SASE^U$ the SASE algorithm augmented with *union* support, which is equivalent to ACEPMatch when $k = 0$ (for queries without negation). Figure 5 shows the throughputs of $Q1$ to $Q4$ of $SASE^U$ and ACEPMatch when $k = 1, 2$, with a window size of 3 million events (about 8 hours). As also found in [18], the throughput decreases significantly for long/large query patterns, which is the case for $Q1$ and $Q4$ (e.g., $Q1$ has 32 letters/L-states). This is because the number of intermediate match nodes (or size of Active Instance Stacks [18]) increases drastically when the number of automaton states is large. In addition, a large window size also makes such intermediate data stay long without expiration (albeit the window pushdown [18]). We also see that, when k increases, the throughput decreases. This is because more non-determinism is added to each match node when k increases, as it needs to be processed and multiplied into more copies at each step through the *propagate* procedure in ACEPMatch.

We then repeat the experiment for a much smaller window size, 100K events, the result of which is shown in Figure 6. While the relative performance among queries is the same, the throughputs in general are much higher. The reason of this is as explained previously for Figure 5—ACEPMatch eagerly checks window expiration and expunges the expired match nodes, corresponding to the *pushing windows down* optimization in [18]. This is very effective in improving efficiency, since otherwise the intermediate match results would multiply quickly with new events.

We then perform this experiment over queries $Q5$ to $Q8$ on the PHEV data with a window size of 50,000 events, and show the result in Figure 7. $Q5$'s pattern has a size of 18, and is the largest among these four queries. In Figure 8, for the same window size as Figure 7, we further examine the throughput as pattern size changes using the PHEV data. To get various sizes, each query is based on the templates $Q5$ to $Q8$, possibly adding/removing a minimum number of letters from a template. Figure 8 clearly shows the trend that as pattern size increases, throughput decreases, because the number of automaton states increases and the chances of event matching and intermediate result propagation grow significantly. We also see that a greater k entails a smaller throughput capability, the reason of which is explained above. In what follows, unless otherwise specified, we use a default value of $k = 2$.

In the next set of experiments, we examine the effectiveness and benefit of RC-ACEP. We build the history $\mathcal{H}$ using ACEPMatch until there are 512 match bundles (i.e., a match map has a size of 512 bits). First, using the Twitter Data, and varying the inter-arrival time between two basic events, we measure the fraction of full matches found by RC-ACEP while fixing $k = 2$ for $Q1$. This is shown in Figure 9. The actual *total number of all matches* is obtained by running ACEPMatch without the inter-arrival time constraints. We compare RC-ACEP's fraction of true matches found with the version of ACEPMatch that has timeouts (i.e., the processing of an incoming letter stops when the next incoming letter arrives). When the inter-arrival time is small, i.e., when the processing engine is overloaded and cannot keep up with the stream rate, RC-ACEP finds a significantly greater fraction of matches than ACEPMatch with timeout. The reason is that RC-ACEP judiciously and efficiently selects the most probable match nodes that may result in full matches in a best-first approach. As the inter-arrival time gets larger, the processing engine starts to catch up, and we see that for a small range, ACEP picks up slightly more matches than RC-ACEP, before they both reach fraction 1. This is because RC-ACEP still has a little overhead in selecting the most promising match nodes in an A*-style search, even though it is small compared to query processing. Both algorithms reach fraction 1 when the processing engine can fully keep up with the data stream.

We do the same experiment with the PHEV Data using a window size of 50,000 events as the default for this dataset. The result on $Q5$ is shown in Figure 10. Again, RC-ACEP finds a much greater fraction of matches than ACEPMatch with timeout when the inter-arrival time is at the low range, although this low range is different from $Q1$, since their *overload* thresholds are different due to query parameters including pattern size and window size. Figure 10 also shows that within a small range ACEPMatch finds slightly more matches than RC-ACEP before they both reach fraction 1 (when they are able to keep up with the stream). The reasons for these are explained earlier. With other queries on the two datasets, we find a similar trend, but at different overload thresholds which depend on factors such as pattern size and window size. In a system involving many patterns

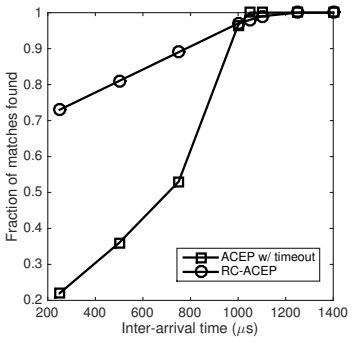


Fig 9 Matches (Twitter)

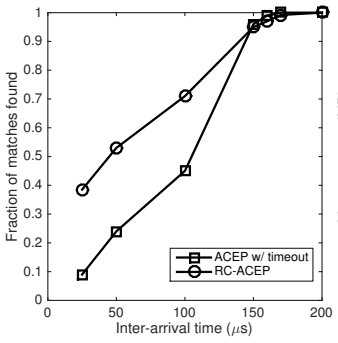


Fig 10 Matches (PHEV)

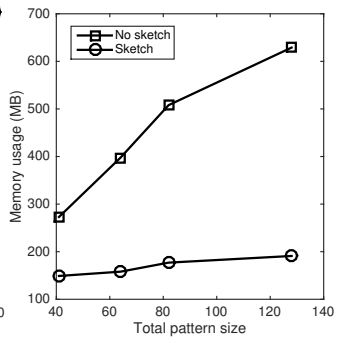


Fig 11 Memory usage (Twitter)

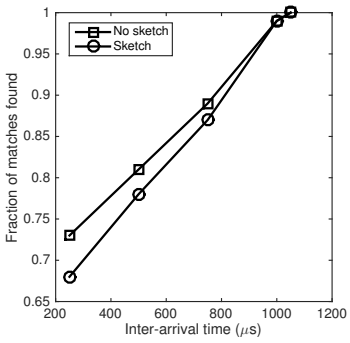


Fig 12 Matches (Twitter)

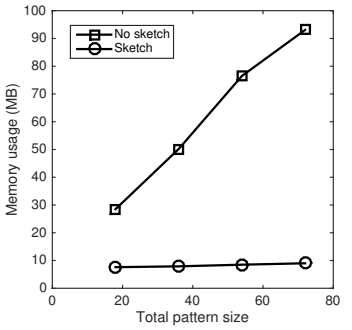


Fig 13 Memory usage (PHEV)

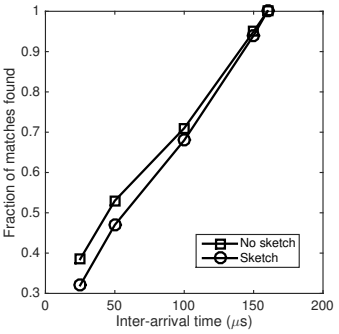


Fig 14 Matches (PHEV)

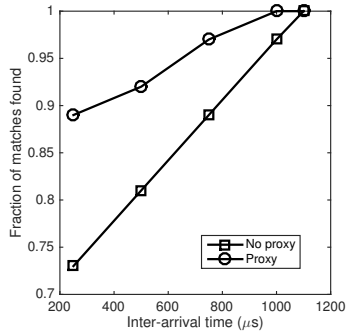


Fig 15 Matches w/ proxy

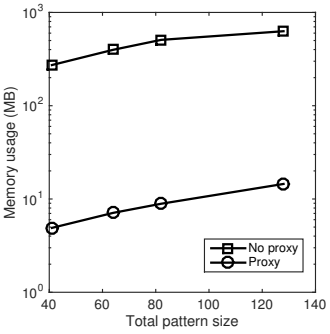


Fig 16 Memory use w/ proxy

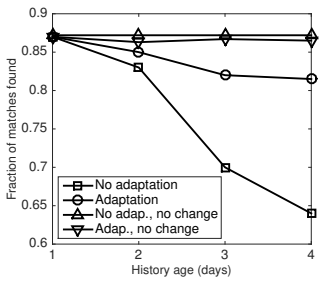


Fig 17 Adaptation vs matches

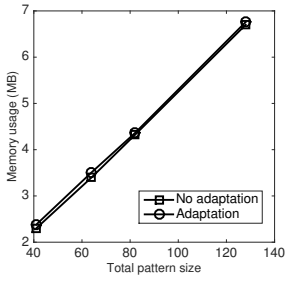


Fig 18 Memory usage

and dynamically changing data streams, the processing engine may be constantly overloaded for any/all queries, and RC-ACEP has clear advantages over ACEPMatch.

We next examine history sketch. We calculate the parameters based on Theorem 4 (while fixing $c = 800$). First, using the Twitter Data, we compare the memory usage of RC-ACEP with and without the sketch, varying the total pattern length (by union of multiple patterns for different companies); the result is in Figure 11. We do the same on the PHEV data; the result is in Figure 13. With the history sketch, it takes significantly less memory. The memory usage of RC-ACEP with sketch grows very slowly, because we always hash the match vectors into the same two-dimensional array. We next measure the fraction of matches found with and without the sketch. As indicated in previous experiments, each query pattern size corresponds to a different stream inter-arrival time threshold. Thus, we show the fraction of found-matches vs. inter-arrival time for query $Q1$ over the Twitter data in Figure 12, and for $Q5$ over the PHEV data in Figure 14 (the results for other queries have a similar trend). We can see that the number of found-matches for the version of using history sketches closely traces that without sketches, and is a little less. The difference is 0 when the processing engine can keep up with the stream rate, because regardless of whether the history sketch causes any error, all match nodes will be processed in time. Using history sketch has some accuracy loss when the system is overloaded mainly because of hash collisions in the data structure. Due to the use of multiple hash functions, this loss is minimized given a compact structure. Moreover, the error is only on one side (i.e., only possibly increasing a count), which is the same for all match nodes; this fact mitigates the accuracy loss as the match-node choice decision is based on the relative comparison among the match nodes.

Next we evaluate the proxy match optimization. Using Twitter Data, we first compare the fractions of full matches found for various event inter-arrival times with and without proxy match for RC-ACEP. The result on $Q1$ is shown in Figure 15. With proxy match, the fraction of full matches found is even higher. This is because there are fewer match nodes in the automaton, which results in greater efficiency. We also measure the memory usage with and without proxy match under various pattern lengths, and show the result in Figure 16. With proxy match, the memory consumed is considerably less. This is because the automaton system keeps significantly fewer match nodes, as discussed in Section 3.3. The results with the PHEV Data show similar trends, and we therefore omit them here.

Finally, we examine the adaptive history update, using Twitter data with the same same settings as in the previous experiment and an inter-arrival time of 300 μs. We use the same length of history as before, and vary the age when the history was built between 1 day and 4 days. In the version of no adaptation, we do not run the Q-learning updates, but only run RC-ACEP itself (with its own updates to $\mathcal{H}$). In the version with adaptation, we run the Q-learning updates presented in Section 5 as well. The result is shown in Figure 17. To examine the performance of Q-learning history

adaptation in the event that there is actually no significant change in data streams, we also run the two versions of the algorithm over a semi-synthetic data stream where we simply duplicate the first day's stream to each of the following days (labeled as "no change" in Figure 17).

First, when data stream has virtually no change, the version using Q-learning history adaptation finds almost as many matches (slightly fewer) as the version without adaptation. The slight difference is due to the small overhead in running Q-learning agents. The overhead is small because each Q-learning agent only follows a single path without branching and forking into multiple paths. However, for the actual data when there are some stream changes over time, the version with adaptation performs much better. The fraction of matches found under the adaptation version remains consistently high, while it drops sharply when the history age is between two and three days. This is because the Q-learning adaptation explores match nodes that are evaluated poorly before but have improved due to stream changes. This exploration result is reinforced through the rewards from match progression. Moreover, this is built into the history structure and corroborated further by the collective decision making in RC-ACEP. In addition, we compare the memory usage between the two versions under different pattern lengths, the result of which is shown in Figure 18. The Q-learning adaptation has little memory usage overhead because a Q-learning agent does not branch out to fork into multiple match trajectories, but only chooses one route.

Summary. Our experiments show that the throughput of ACEP decreases for large patterns or when the error threshold or window size increases. The system can be easily overloaded when monitoring many queries. To keep up with the stream rates, it is necessary to use RC-ACEP, which is effective in judiciously selecting the most promising match nodes to process. Using history sketch has a small and nearly constant memory footprint with little loss on the fraction of full matches. The proxy match optimization is very effective in further reducing the memory foot-print as well as in improving the fraction of matches found. Finally, the adaptive history building is helpful in keeping the history structure up-to-date and incurs little memory overhead.

7. OTHER RELATED WORK

Most closely related work is discussed in its context above. In Section 2, we have discussed at length the connections between ACEP and specific CEP languages and systems. In addition, Zhao and Wu [19] study approximate event processing in a content-based publish/subscribe system, and propose a hierarchical indexing table to store subscriptions and to give approximate answers. However, their event queries are only limited to range queries on an attribute; i.e., they do not deal with complex event queries.

To our knowledge, previous work does not consider the resource-constrained processing of complex event queries. Sketch is a probabilistic technique that has been applied to data streams as well as approximate query processing [8]. However, we modify a count-min sketch significantly to include both counts and bitmaps for our purpose of sketching the history. Further more, our changes require a novel analysis on the probabilistic properties and the choice of parameters. Finally, Q-learning [17] is a model-free reinforcement learning technique. Mapping our ACEP matching problem to Q-learning states and actions is our contribution with the goal of dynamically updating the history structure.

8. CONCLUSIONS

In this paper, we propose the ACEP algorithm and a novel RC-ACEP algorithm based on collective learning from a history data structure. We prove the competitive ratio of our online algorithm, which is optimal in the domain of polynomial algorithms. Moreover, we devise the proxy match optimization and a history sketch with its analysis. Finally, we devise a scheme to dynamically update the history structure with little overhead.

Acknowledgment. This work was supported in part by the NSF, under the grants IIS-1149417, IIS-1319600, and IIS-1633271.

9. REFERENCES

[1] `https://dev.twitter.com/streaming/public`.
[2] `https://en.wikipedia.org/wiki/Most_common_words_in_English`.
[3] `http://www.cs.uml.edu/~ge/paper/rc_acep_tech_report.pdf`.
[4] `http://www.ee.ucr.edu/~hamed/PEVData.zip`.
[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.
[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.
[7] H. Akhavan-Hejazi, H. Mohsenian-Rad, and A. Nejat. Developing a test data set for electric vehicle applications in smart grid research. In *VTC*, 2014.
[8] G. Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in DB*, 2011.
[9] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 2012.
[10] A. Damiano et al. Vehicle to grid technology: state of the art and future scenarios. *J. of Energy & Power Engineering*, 2014.
[11] U. Feige. A threshold of ln n for approximating set cover. *JACM*, 1998.
[12] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE TSSC*, 1968.
[13] Z. Li and T. Ge. PIE: Approximate interleaving event matching over sequences. In *ICDE*, 2015.
[14] Y. Mei and S. Madden. ZStream: A cost-based query processor for adaptively detecting composite events. In *SIGMOD*, 2009.
[15] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. In *SIGMOD*, 2012.
[16] E. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of math. biology*, 1989.
[17] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 2010.
[18] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.
[19] Y. Zhao and J. Wu. Towards approximate event processing in a large-scale content-based network. In *ICDCS*, 2011.