AYAC: A Probabilistic Complex Event Processor for Incomplete Streams[*]

Mohammad Toossi
toossi@umd.edu

University of Maryland, College Park

Abstract. In this paper, we present the design of AYAC, a data stream processor designed to efficiently support complex event queries defined over an unreliable stream of simple events. Each complex event is specified using a SQL-style query over a sequence of events in a fixed order. Negative events, events that should not occur in the stream, are also supported. A probability is assigned to losing each event in the input stream. AYAC can detect complex events with missing components. The probability that the complex event has actually happened is estimated and reported using a model based on the past history of the input stream. Using several optimizations allows our system to evaluate each input event in time linear in total number of query states, and consume memory linear in query time window size.

1 Introduction

As receptor devices such as RFID readers and various wireless sensors become cheaper, a quickly growing number of businesses deploy large networks of these devices. Applications then need to execute complex business logic on top of data streams produces by those networks[11]. The business logic is usually defined in terms of complex event patterns detected in the data stream.

The main component of such applications is usually a system that receives the data stream as input and outputs the triggered complex event patterns. This component is called a *Complex Event Processor (CEP)* or an *Event Stream Processor (ESP)*.

Typically, input stream is assumed to consist of a uniform sequence of *Simple Events* received in real time as events are sensed by receptor devices. Each simple event at least includes a timestamp and the identifier of the receptor. Additionally, a set of queries or *Complex Events* is registered with the CEP.

Depending on the type of events and application, the data stream throughput, the number of queries, and the complexity of the query can vary significantly. For example, in a publisher/subscriber system, new simple events arrive at a slow rate, and the queries are relatively trivial, but the number of queries is very high.

[*] Submitted as scholarly paper for M.Sc. degree in Computer Science; Spring 2007

In this paper we focus on networks of RFID receptors. In such networks, a relatively high-throughput data stream is generated, and queries have moderate complexity. However, the total number of queries is usually limited. Efficient time and memory management are some of the primary concerns of a real-time RFID stream processor.

One aspect of RFID networks mostly ignored by previous work is the inaccuracy of RFID sensors. A sensor may read an RFID tag partially or incorrectly, or not read it at all. A traditional CEP completely misses a complex event even if one of its simple event components is not read accurately.

This is our motivation for designing AYAC, a probabilistic event processor for inaccurate data streams. AYAC attempts to efficiently estimate the probability that each complex event is triggered at any given time. If this probability is above a user-defined threshold, the complex event is output along with its estimated probability.

Next subsection provides a quick survey of related previous work. In Section 2, we define simple and complex events more precisely and outline the features of our complex event specification language. Section 3, explains how event probabilities are computed for a given input stream using our probabilistic model. Then, we illustrate the basic concepts behind our NFA-based event detection algorithms in Section 4. Next, Section 5 summarizes some of the critical optimizations that make our algorithms practical for high-throughput streams. Conclusion and future work are discussed in Section 6.

1.1 Related Work

There has been extensive research on sequence query processing and complex event processing (CEP), both on the definition of the query languages and on the efficient implementation of these languages. This is because this work applies to a very wide range of applications from RSS feeds to database maintenance tasks to intrusion detection. On the other hand, noisy data streams are only a problem in a few cases. This fact along with the novelty of the area has limited the amount of work done on probabilistic version of sequence query processing.

In publish/subscribe (pub/sub) systems[1][3][7], there are generally many subscribers, each interested in a subset of the published events specified by a predicate-based filter. The goal here is to efficiently match each new event against a high number of predicates. Each predicate, however, is stateless and operates on one event at a time.

A recently proposed more expressive pub/sub system[6] suggests stateful subscriptions, effectively allowing predicates spanning multiple events. However, the goal is still efficiently handling many subscribers. This makes the NFA-based implementation impractical for some of the scenarios we are dealing with. Additionally, queries requiring absence of events are not well supported.

The research on active databases [15][5][4][8] is also closely related to CEP over streams. An active database (as opposed to a passive one) allows the users to define persistent rules. These rules consist of a condition over in-database events and an action that gets triggered as soon as the condition is satisfied. Conditions are written based on simple database events such as inserts or complex events specified using event processing algebras over event histories.

COMPOSE from the Ode system[8] and the SNOOP[4] composite event language are examples of two such active database query processing languages. They define operators such as as conjunction, disjunction, negation and sequence on the event history. There is a high number of proposed event algebras with various inconsistencies and peculiarities.

Zimmer[15] analyzes the foundation of complex events and provides a metamodel for any event algebra, decomposing them along multiple independent dimensions. The main issue with active database query languages is that they do not support sliding time windows, and they generally do not allow queries to include constraints that compare values of simple events being matched to each other.

The work on sequence databases is also related. Unlike traditional databases that store unordered tuples, a sequential database is aware of the data ordering and is capable of performing sequence oriented queries more efficiently. Queries are generally specified by extensions of the SQL language. AQuery[9], SQL-TS[12], and SEQ[13] are three such systems. Since queries are still processed similar to traditional databases (e.g. joining relations), they are too slow for our application. Also, support for non-occurrence queries is very limited.

SASE[14] defines a stream CEP language by further enhancing these query languages. A native sequence operator and allowing for more complex non-occurrence constraints are some of its main features. The NFA-based implementation includes a number of optimizations to achieve acceptable speed and memory usage. As a result, SASE seems like a practical system to be used over high-throughput data streams. AYAC is most closely related to SASE. Our query language and processing methods are derived from SASE, but have been heavily enhanced and adjusted to the probabilistic problem at hand.

Barga et al.[2] attempt to unify all the different query processing languages and pub/sub systems. They argue that these systems differ only in the target workload, language features and consistency requirements. Moreover, they propose CEDR and formally define a spectrum of consistency levels. This effort seems to be currently concentrated more on theoretical aspects of this new language and efficient implementation has yet to be worked on.

The vast majority of previous work on sequence query processing deals with precise data streams. Rizvi[10] introduces the concept of probabilistic complex event processing (PCEP) after providing a comprehensive survey of the area. He uses a state tracking module which interacts with CEP and stores the current probabilities for various events in the system. Incoming simple events also have probability values associated with them. PCEP is introduced as a general framework without any concrete practical details.

In case of RFID events, it is not acceptable to assume we have a probability or confidence value associated with each simple event. Generally, enough redundancy is available in RFID information to determine whether a particular RFID read is bogus or not, and to either correct or discard the given read accordingly resulting in an incomplete but accurate data stream.

2 Simple and Complex Events

In this section we define our simple events more precisely, and describe the structure of our complex events as well as a language for specifying them.

2.1 Simple Events

AYAC receives a real-time stream of simple events as input. A simple event is generated whenever an RFID tag is detected by an RFID reader. The simple event contains basic information about the read such as timestamp, reader ID, and the tag contents. For simplicity, we assume that the simple event is a k-tuple that also contains all the extra information relevant to this tag that may be used in a query. In practice, this information is typically obtained by querying a database using RFID tag contents as the key.

For example, in a library, an RFID tag attached to a book may be scanned and then searched in a database to find the book title and category. However, we assume that simple events received by AYAC already contain book title and category which may be referenced in queries.

Furthermore, we assume that this uniform stream of k-tuples is delivered to AYAC in the order of increasing timestamps and that it is error free. That is, an external buffering mechanism is used to hold tuples that arrive out of order, and all partial or noisy reads are completely dropped from the stream (e.g. using a check-sum).

2.2 Complex Events

We define a complex event to be a particular set of simple events occurring in a given order within a time window. A complex event can also specify certain simple events that should not appear in the sequence at given positions. AYAC processes the incoming stream of simple event tuples and outputs the complex events that could have been triggered by these tuples with a probability higher than a threshold set by the complex event. The output includes contents of all tuples triggering the event along with the probability of the event.

We use shop lifting detection in a retail store[11][14] as our example: An RFID tag is attached to every item in the store. RFID receptor A is installed above the shelves, receptor B is installed above the checkout counter, and receptor C is installed at the exit. A shop lifting is detected if a tag ID is read at A and C without being read at B in between.

Complex events in AYAC are specified in a format based on the SASE[14] event language with a few enhancements and adjustments to accommodate our more general assumptions. Instead of the formal definition, we illustrate the main features of our event language by defining the shoplifting event and looking at each component here:

```
SEQUENCE (x,!y,z)
WHERE (x.receptor['A'] = 'A') AND
      (y.receptor['B'] = 'B') AND
      (z.receptor['C'] = 'C') AND
      (x.id[y.id] = y.id[x.id]) AND
      (y.id[z.id] = z.id[y.id])
WITHIN 12 hours
THRESHOLD 0.80
```

SEQUENCE Clause All complex events fix the ordering of underlying simple events in time. The `SEQUENCE` clause simply assigns a name to each simple event so that they can be referenced in the `WHERE` clause.

It also specifies the event type: A *Positive Component*, which appears without a leading "`!`" in the clause, is a simple event that should exist in the sequence. On the other hand, a *Negative Component*, specified with "`!`", should not appear anywhere in the stream between the preceding and succeeding positive components in the sequence.

We note that "ANY" operator defined by the SASE event language is no longer necessary due to our generalized format. Since simple events are untyped, an entire "ANY" clause can be replaced with a single component here.

WHERE Clause Similar to an SQL query, this clause defines predicates on attributes of the positive and negative components labeled in the `SEQUENCE` clause. However, since some of the events may be missing when the query is being evaluated, a default value is specified for each attribute in brackets. For example, if AYAC believes that event `x` has occurred but missed by the sensor, the fourth predicate still evaluates to true because it will be translated to `y.id = y.id`.

The `WHERE` clause is always converted to disjunctive normal form (DNF). We define a "predicate" as a conjunctive clause in the DNF. We refer to a predicate as a *Trivial Predicate* if it only depends on the values of a single component. In our example, the first three predicates are trivial while the last two are not.

As we will see in the next sections, trivial predicates play a critical role in event detection. As a result, we require that, for each sequence component, at least one trivial predicate is present. A particularly common class of trivial predicates are those that fix the RFID reader of a component.

WITHIN Clause The `WITHIN` clause specifies a time window length. The entire event sequence should happen within the time window. This allows us to discard any events older than time window length. Additionally, a negative component at the very beginning or very end of a sequence only becomes meaningful when a time window is present.

THRESHOLD Clause This is the minimum probability required for the event to be reported. AYAC drops partially and fully matching events with lower probability. Setting this threshold to a very low value can degrade AYAC's performance even if not many matches are produced because of the internal optimizations performed using the threshold. Usually, threshold should be higher than ϵ^2. (ϵ is the error rate of the RFID reader as defined in Section 3.)

3 Probabilistic Model

The simple event stream received by AYAC is incomplete as a result of RFID tags being missed by RFID readers. A model for estimating the probability that a given event occurred but was lost is needed. We estimate these probabilities by gathering statistical information from previously observed events, and by making some basic probabilistic assumptions.

Our first assumption is roughly the *Markov property*: The timing of component e_i in a complex event sequence $(e_1, e_2, \cdots, e_n)$ only depends on the time that the last positive component preceding e_i happens. (i.e. e_{i-1} if it is not negated, otherwise e_{i-2} if it is not negated, and so on.) For example, the time difference between e_i and e_{i-1} does not depend on when e_1 occurred. This is a reasonable assumption because the sequence usually tracks the physical location of one or more RFID tags as they move, and the time we observed them at last location is the most significant factor in predicting the time we will observe them at next location. The second assumption is that all instances of a complex event behave similarly.

Therefore, we can define probability density function $f_i(t)$ that e_i happens t time units after the previous positive component. Estimating this function is a separate task which can be performed in the background using history information gathered from previously detected complex events. Alternatively, a simple probabilistic model may be used based on the event semantics. This process does not have any other interaction with the main event processor and is not discussed any further in this paper.

Our next assumption is that an RFID reader misses or drops a tuple with fixed probability ϵ. This can be easily generalized to assign a different probability to each component of a complex event (and usually each RFID reader). Finally, we define $F_i(T)$ to be the cumulative distribution function $F_i(T) = \int_0^T f_i(t)dt$ for later use.

In the next few subsections, we show how complex event probabilities can be computed using our model in various situations.

3.1 Missing A Positive Event

Let E_1 be a complex event defined by sequence $(e_1, e_2, \cdots, e_n)$. However, we have observed $(e_1, e_2, \cdots, e_{i-1}, e_{i+1}, \cdots, e_n)$ in the incoming stream. We need to compute the probability that E_1 has happened. This is the probability that e_i happened between e_{i-1} and e_{i+1} but was missed by the RFID reader. Let T be the time gap between e_{i-1} and e_{i+1}. Given $f_i(t)$, the probability density of e_i happening t time units after e_{i-1}, we have:

$$P(E_1 \text{ has happened}) = \epsilon \int_0^T f_i(t)dt = \epsilon F_i(T) = S_i^P(T)$$

The above result, named S_i^P, is a non-decreasing function of T specifying the probability of missing positive event e_i in a time interval of length T.

3.2 Missing A Negative Event

This time, we are given complex event E_2 specified by sequence $(e_1, \cdots, e_{i-1}, !e_i, e_{i+1}, \cdots, e_n)$, and observed sequence $(e_1, \cdots, e_{i-1}, e_{i+1}, \cdots, e_n)$. The probability that E_2 has really happened should be computed because there is a chance that negative event e_i has happened but was missed by the reader. Similar to the previous case, the probability that e_i happens between e_{i-1} and e_{i+1} with time gap T is $F_i(T)$. Therefore, we have:

$$\begin{aligned}
P(e_i \text{ does not happen}) &= 1 - F_i(T) \\
P(e_i \text{ is not observed}) &= P(e_i \text{ does not happen}) + P(e_i \text{ is missed}) \\
&= (1 - F_i(T)) + \epsilon F_i(T) \\
P(E_2 \text{ has happened}) &= P(e_i \text{ does not happen} \mid e_i \text{ is not observed}) \\
&= \frac{P([e_i \text{ does not happen}] \wedge [e_i \text{ is not observed}])}{P(e_i \text{ is not observed})} \\
&= \frac{P(e_i \text{ does not happen})}{P(e_i \text{ is not observed})} \\
&= \frac{1 - F_i(T)}{\epsilon F_i(T) + 1 - F_i(T)} \\
&= S_i^N(T)
\end{aligned}$$

Note that the function S_i^N for missing negative event e_i in time T is non-increasing.

3.3 Multiple Missing Events

In the general case, missing tuples in complex event E can be an arbitrary subset of the event sequence $(e_1, \cdots, e_n)$. However, if the subset consists of multiple non-adjacent segments in the sequence, the probability can be independently computed for individual segments and multiplied together. Furthermore, if more than one positive event is missing, the resulting probability will be below ϵ^2 which can be assumed to be much lower than threshold for majority of applications. Below, we compute probabilities for the most significant cases:

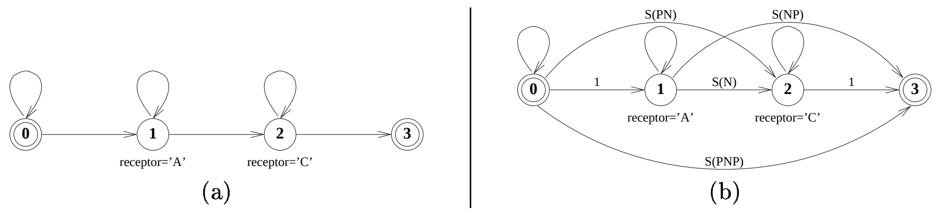

Fig. 1. NFA states for the shoplifting example. *Initial and final states are indicated with double circles. Trivial predicates available for internal states are written below the state. (a) shows the basic precise model for the complex event. (b) is the imprecise version where each edge is labeled with a scoring function from Section 3. $S(PN)$ refers to S^{PN}, $S(N)$ refers to S^{N}, and so on.*

Two Positive Events Two consecutive events e_i and e_{i+1} have happened in time interval T but both have been missed. For every t in $(0, T)$, we integrate the probability of e_{i+1} happening at t and e_i happening in $(0, t)$:

$$S_i^{PP}(T) = \epsilon^2 \int_0^T f_{i+1}(t) F_i(t) dt$$

Multiple Negative Events The probability that negative events $e_i, e_{i+1}, \cdots, e_j$ have not happened in time interval T assuming that they have not been observed, can simply be computed by multiplication because of our independence assumption:

$$S_i^{N \cdots N}(T) = \prod_{k=i}^{j} S_k^N(T)$$

A Positive Event Surrounded By Negative Events For missing sequence $!e_{i-p}, \cdots, !e_{i-1}, e_i, !e_{i+1}, \cdots, !e_{i+q}$, we use a combination of the techniques used above to get:

$$S_i^{N \cdots NPN \cdots N}(T) = \epsilon \int_0^T \left(\prod_{k=1}^{p} S_{i-k}^N(T) \right) f_i(t) \left(\prod_{k=1}^{q} S_{i+k}^N(T) \right) dt$$

4 Basic Event Processor

A complex event can be modeled as a *Non-deterministic Finite Automata (NFA)* such as in Figure 1(a). States are named $V_0, V_1, \cdots, V_l, V_{l+1}$, where V_0 is the initial state, and V_{l+1} is the final state. NFA starts at V_0 and can make a transition per input tuple. However, it can only move to a connected state if the tuple satisfies the predicates in that state. Also, it can always move to V_{l+1} if it is reachable. Note that NFA does not have any states corresponding to the negative components of the complex event, and only checks trivial predicates. Negative components and non-trivial predicates are checked at a later stage in the algorithm.

To accommodate the lossy input stream, we allow the NFA to skip states which would have been matched by dropped tuples. However, the likelihood of such events happening has to be taken into account. We label each edge with a score corresponding to the probability that, given the current input, this transition represents the actual sequence of events (Figure 1(b)). We define a path's score as the multiplication of all scores on its edges. Upon reaching the final state, path score (which equals event probability) is computed and compared to threshold.

More precisely, edges are labeled with scoring functions such as the ones shown in Section 3 since the score is a function of the amount of time passed between the two states. We refer to scoring function from V_i to V_j as $H_{i,j}(t)$. Also, we define $G_{i,j}(T) = \max_{t\in[0,T]} H_{i,j}(t)$. Both functions should be very efficiently computable (e.g. using lookup tables). Finally, we note that many edges can be removed because $G_{i,j}(W)$ is below threshold, where W is the query time window length.

AYAC needs to efficiently keep track of states reached by the NFA (and their partial scores) as it feeds input tuples to it. This is achieved using *Probabilistic Arrival Queues (PAQs)* which are enhanced versions of *Active Instance Stacks[14].*

4.1 Probabilistic Arrival Queue (PAQ)

At any moment through the lifetime of a query, its NFA can be in multiple states, having arrived to each one via several different paths. This information is efficiently tracked using data structures called PAQs. A PAQ is created for each state of the NFA except the initial and final states.

First, we observe that we can identify a path uniquely using an $(l+2)$-tuple $(a_0, a_1, \cdots, a_l, a_{l+1})$ where a_i is the initial *arrival* time of this path at state V_i. This is possible since NFA does not have any cycles except the self-loops. Still, explicitly storing arrival timestamps for every traversed path is impractical. As a result, we use a PAQ per state. A PAQ simply stores all arrival times (and some other information) to that state without regard to the possibly many paths that may have arrived at that state at that time. When NFA eventually reaches the final state, we can use PAQs to reconstruct all paths and evaluate them more thoroughly.

Each PAQ is a queue of (t, u) pairs where t is the arrival time, and u is an upper bound to the maximum score of all paths reaching the state at t. u is used by AYAC to significantly reduce memory and time requirements by early elimination of paths that already have a low score.

Enqueue Operation Whenever a new input tuple is received, we need to determine if it can cause a new arrival (a transition that is not a self-loop).

For each state V_i, the tuple is evaluated against trivial predicates specified by that state. Only if the tuple satisfies all predicates, NFA may go to V_i[1]. In this case, let $(t_{j,k}, u_{j,k})$ be the k-th tuple in PAQ of state V_j. Path's reaching V_j have maximum score $u_{j,k}$ and they can move to V_i using an edge with scoring function $H_{j,i}$. Let t_{new} be the timestamp of the new tuple. The upper bound on scores of all paths arriving in V_i at t_{new} can be computed as:

$$u_{\text{new}} = \max_{j \in \{1, \cdots, i-1\}, k} u_{j,k} \times H_{j,i}(t_{\text{new}} - t_{j,k})$$

We enqueue $(t_{\text{new}}, u_{\text{new}})$ only if u_{new} is above threshold. Arrivals with lower scores can be eliminated because a path's score cannot increase as it extends.

Unfortunately, evaluating every tuple in PAQs of every state preceding V_i is an expensive operation. Dealing with this issue is one of the main focuses of Section 5.

Dequeue Operation Any arrival before $t_{\text{new}} - W$ can be safely discarded. Since pairs in a PAQ are ordered by increasing t, this amounts to simply dequeuing old tuples. Dequeue operation ensures that amount of memory consumed by PAQs is in $O(W)$ and does not grow indefinitely.

Removing tuples may enable us to improve (i.e. decrease) upper bounds in the existing pairs in succeeding PAQs (possibly as a background process). However, the naive approach for this operation would be too time consuming. We leave efficient upper bound adjustments as future work. Of course, future arrivals still benefit from dequeue operation because u_{new} is computed using only recent arrivals.

Initial and Final States So far we have ignored transitions from V_0 and transitions to V_{l+1}. There are no PAQs associated with these states. Final transitions are discussed in the next subsection when we explain finalizing event matches.

An initial transition from V_0 to a state V_i that matches the input tuple can happen at any time. The exact score for the transition cannot be determined until the final state is reached. Therefore, our new upper bound u'_{new} considers initial transition with every possible time distance up to W:

$$u'_{\text{new}} = \max \left\{ u_{\text{new}}, \max_{t \in [0,W]} H_{0,i}(t) \right\} = \max \{ u_{\text{new}}, G_{0,i}(W) \}$$

4.2 Match Generation

In this paper, we only explain detection of complex events ending with a positive component when that component is not missing in the input. Some of the other cases (e.g. when initial component is positive and not missing but the ending is

[1] For queries with long sequences, usually an index can be used to quickly locate matching states.

arbitrary) can be detected similarly, but the general case is overly complicated and even the query semantics have to be defined using non-intuitive assumptions.

In this case, the final transition to V_{l+1} always happens as soon as V_l is reached. Moreover, it does not affect the score because $H_{l,l+1}(t) = 1$. Consequently, AYAC initiates match generation as soon as a new pair $(t_{\text{new}}, u_{\text{new}})$ is added to PAQ for V_l. During this process, AYAC considers all the paths leading arriving in V_{l+1} at t_{new} and filters them using non-trivial predicates of the query, the time window constraint, and negative events that may have occurred. The remaining results are output as matching complex events.

Naive Approach Take $(l + 2)$-tuple $(a_0, a_1, \cdots, a_l, a_{l+1})$ containing arrival timestamps that uniquely identifies a path from V_0 to V_{l+1}. If such a path exists, clearly we can find timestamp a_i in PAQ for V_i. Also, $a_0 \leq a_1 < \cdots < a_l \leq a_{l+1}$.

Using this result, the naive algorithm enumerates all possible paths by choosing one element from each of the PAQs for V_1 to V_{l-1} with timestamps t_1 to t_{l-1} respectively, such that $t_{\text{new}} - W \leq t_1 < t_2 < \cdots < t_{l-1} < t_{\text{new}}$[2]

Next, simple events with these timestamps are retrieved. These events have already satisfied trivial predicates and are guaranteed to fit in the time window. The rest of the predicates are applied to these tuples, and if successful, negative components of the query are searched for in the appropriate intervals of the stream.

Enhanced Approach To speedup detection of negative events occurring in a the path, AYAC maintains a *Negative Arrival Queue (NAQ)* for each negative component. Similar to PAQs, NAQs queue the timestamps of tuples matching trivial predicates of the negative component, but they are much faster to maintain because they only hold timestamps (and no upper bounds). Searching for negative events in a time interval can then be efficiently performed using a binary search in the NAQ.

More importantly, paths are iteratively extended by starting at V_l and going backwards. Every predicate applicable to this portion of the path is applied, and negative tuples are searched for before extending the path. Also, the partial score of the path combined with the upper bound for the rest of the path helps us eliminate many paths in advance.

More precisely, if the current partial path goes from (t_b, u_b) in V_j to (t_c, u_c) in V_l and has score s, we can extend it to (t_a, u_a) in V_i only if all of the following conditions hold:

- A transition is available to the beginning of the partial path in V_j: $i < j \wedge t_a < t_b$
- Upper bound on the score for the completed path is above threshold. This upper bound is computed as: $s \times H_i, j(t_b - t_a) \times u_a$

[2] Note that the timestamp for arriving in V_0 is $t_{\text{new}} - W$. This is to allow maximum space for matching initial negative components.

– No negative elements are found between t_a and t_b. This is done by retrieving an initial list of elements that match trivial predicates using NAQ. The path can be eliminated if we can evaluate every predicate involving the negative component just by using the current partial path.
– Applying additional (non-trivial) predicates involving (t_a, u_a) is successful. In other words, non-trivial predicates are applied to the path as soon as all the required components are matched.

5 Optimizations

In this section, we very briefly outline the two class of optimizations that are critical for real-time processing of high-throughput streams.

5.1 Upper bound Computation

Obtaining reasonable upper bounds for PAQ entries is crucial for eliminating the huge number of unnecessary paths generated by traversing low scoring edges of the NFA. However, directly computing the upper bound formula given in subsection 4.1 is expensive and impractical. AYAC employs several optimizations to compute looser but faster upper bounds.

For instance, upper bound v_{new} is derived as follows:

$$\begin{aligned} u_{\text{new}} &= \max_{j\in\{1,\cdots,i-1\}} \max_k \left(u_{j,k} \times H_{j,i}\left(t_{\text{new}} - t_{j,k}\right)\right) \\ &\leq \max_{j\in\{1,\cdots,i-1\}} \left[\left(\max_k u_{j,k}\right)\left(\max_k H_{j,i}(t_{\text{new}} - t_{j,k})\right)\right] \\ &\leq \max_{j\in\{1,\cdots,i-1\}} \left[\left(\max_k u_{j,k}\right) \times G_{j,i}(t_{\text{new}} - t_{j,0})\right] \\ &= v_{\text{new}} \end{aligned}$$

The key to evaluating v_{new} quickly is being able to compute $\max_k u_{j,k}$, the highest upper bound stored in a given PAQ at any time, efficiently. This can be done in amortized $O(1)$ time by augmenting the PAQ with auxiliary information as explained in the extended version of this paper. This reduces the running time of u_{new} computation to $O(l)$. As a result, the enqueue operatoin is $O(l^2)$ in the worst case. This can be reduced to $O(l)$ assuming that threshold is higher than ϵ^2.

Furthermore, using addition data structures not discussed here, AYAC can significantly improve v_{new} (sometimes even reaching u_{new}) whenever a monotonic scoring function is used. As seen in Section 3, the vast majority of scoring functions are monotonic.

5.2 Hashed PAQs

Very frequently, a complex event requires at least one particular attribute of tuples, such as the RFID tag, to be equal across all components. (See PAIS in [14].)

For such queries, AYAC can perform enqueue operations and match generation much more efficiently. Elements of the PAQ are partitioned by a hash function computed on all equality attributes, and a linked list is maintained for elements in each partition. Again, more details are available in the extended version of the paper.

6 Conclusion and Future Work

We provided design and implementation suggestions for a probabilistic complex event processor. An NFA is defined for each complex event. Each NFA then processes input tuples as they arrive and triggers the complex event in real-time as soon as the last component is detected. Inputs dropped or missed by RFID readers lower the event's likelihood but still trigger it if the likelihood is above the threshold.

Using our optimizations, processing time per tuple per complex event per state is in amortized $O(1)$ time. Maximum memory used for each event is linear in the time window length but is frequently much lower. Trivial predicates and equality constraints play an important role in the real-world performance and memory usage of the system.

Several assumptions which were made to simplify the problem can be basis for further research. For instance, converting the `WHERE` clause to disjunctive normal form and then requiring trivial predicates with high selectivity may prove too restrictive for some applications. Also, complex events with a negative last component or missing positive last component are also currently being ignored.

Moreover, the performance may be improved by finding efficient methods for updating PAQ upper bounds as old elements are discarded in preceding PAQs. Currently, discarding elements only improves upper bounds computed for tuples inserted in the future.

References

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.
2. R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.
3. A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 163–174, New York, NY, USA, 2003. ACM Press.
4. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

5. U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.
6. A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.
7. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):115–126, 2001.
8. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.
9. A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.
10. S. Rizvi. Complex event processing beyond active databases: Streams and uncertainties. Master's thesis, EECS Department, University of California, Berkeley, December 16 2005.
11. S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 885–887, New York, NY, USA, 2005. ACM Press.
12. R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.
13. P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *The VLDB Journal*, pages 99–110, 1996.
14. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM Press.
15. D. Zimmer. On the semantics of complex events in active database management systems. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 392, Washington, DC, USA, 1999. IEEE Computer Society.