High-Performance Dynamic Pattern Matching over Disordered Streams

Badrish Chandramouli[†], Jonathan Goldstein[†], David Maier[‡]
[†]Microsoft Research, Redmond, Washington, USA
[‡]Portland State University, Portland, Oregon, USA
{badrishc, jongold}@microsoft.com, maier@cs.pdx.edu

ABSTRACT

Current pattern-detection proposals for streaming data recognize the need to move beyond a simple regular-expression model over strictly ordered input. We continue in this direction, relaxing restrictions present in some models, removing the requirement for ordered input, and permitting stream revisions (modification of prior events). Further, recognizing that patterns of interest in modern applications may change frequently over the lifetime of a query, we support updating of a pattern specification without blocking input or restarting the operator. Our new pattern operator (called AFA) is a streaming adaptation of a non-deterministic finite automaton (NFA) where additional schema-based user-defined information, called a *register*, is accessible to NFA transitions during execution. AFAs support dynamic patterns, where the pattern itself can change over time. We propose clean order-agnostic pattern-detection semantics for AFAs, with new algorithms that allow a very efficient implementation, while retaining significant expressiveness and supporting native handling of out-of-order input, stream revisions, dynamic patterns, and several optimizations. Experiments on Microsoft StreamInsight show that we achieve event rates of more than 200K events/sec (up to 5× better than simpler schemes). Our dynamic patterns give up to orders-of-magnitude better throughput than solutions such as operator restart, and our other optimizations are very effective, incurring low memory and latency.

1 Introduction

The advent of the Digital Age has made large-scale data acquisition and online processing a crucial component of modern systems. A *Data Stream Management System* (*DSMS*) [1, 4, 22, 25] enables applications to issue long-running *continuous queries* (*CQs*) that efficiently monitor and process streams of data in real time. DSMSs are used for data processing in a broad range of applications including clickstream analysis, fraud detection, and algorithmic trading of stocks. Recently, *pattern CQs* — where the user wants to detect patterns across time — have garnered significant attention in the research community [2, 10, 16, 17, 19, 28].

New Challenges Modern business needs pose several new challenges for pattern CQs, that must be addressed *in entirety* for a deployable solution. The following example illustrates some of the challenges that practical solutions for pattern CQs must confront.

Example 1 (Algorithmic Trading)**.** *An automated stock-trading application wishes to use a DSMS to perform technical analysis to detect interesting* chart patterns *[8] in real-time. Here, each event in the stream contains stock data such as symbol, price, price change, and volume. A simple example is the* V-pattern*, where we look for consecutive stock events that consist of downticks followed by upticks. Other examples of chart patterns include wedge, double top, and candlestick.*

A pattern CQ, call it Q_1, may wish to detect a sudden large price drop (of say δ), followed by a sequence of k consecutive V-pattern occurrences, such that the total number of upticks is equal to the total number of downticks (across the k V-patterns). Here, k is large and may not be known in advance. The stopping condition could be data-dependent — a second pattern CQ, called Q_2, may wish to report the number of consecutive V-patterns (after a large price drop) until the price reaches the original price before the drop.

1) Execution Model Patterns over streams can involve unbounded looping operations such as Kleene closure and are not expressible using standard DSMS operators such as joins and windows. One execution model for patterns CQs is a *non-deterministic finite automaton* (NFA) to match regular expressions [16, 19]. However, many patterns such as the V-pattern above are not regular — we need to associate additional information with computations, e.g., a count of upticks and downticks encountered during matching.

Recent research [2, 10, 20] recognizes this limitation of plain NFAs and enhances expressiveness by allowing added runtime information to be associated with NFA states — we refer to these automata as *augmented* NFAs. These proposals *constrain* the transitions allowed in an NFA; specifically, the NFA is a linear chain of states, with transitions from state i only going to states i and $i + 1$.

We continue development in this direction by allowing *unconstrained* augmented NFAs with arbitrary transitions; we will see in Section 3 that complex patterns such as Q_1 and Q_2 above are easier to describe and execute efficiently using an unconstrained model. The challenge is to support this model efficiently, with robustness in the face of real stream characteristics, such as high volume, out-of-order events, and modifications to previous events.

2) Dynamic Patterns The value proposition of a DSMS to a business is captured by the *monitor-manage-mine* (*M3*) loop in Figure 1. We *monitor* data in real-time to detect interesting patterns, that are used to *manage* and perform business actions. The raw data is aggregated and stored offline; we *mine* the historical data to determine new patterns (or modifications to existing patterns) that are fed back to the monitor phase. In the trading scenario of Example 1, historical data mining can reveal new chart patterns or refinements

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

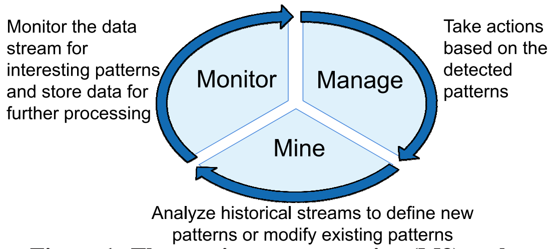

Monitor the data stream for interesting patterns and store data for further processing
Monitor
Manage
Take actions based on the detected patterns
Mine
Analyze historical streams to define new patterns or modify existing patterns

Figure 1: The monitor-manage-mine (M3) cycle.

of existing patterns (e.g., changed optimal value of k), which need to be deployed in real-time to trigger automatic trading actions.

Traditionally, the mining phase to determine pattern changes has been largely human-driven and infrequent. However, there is an increasing need [14] to adjust quickly to fast-changing trends by completing the M3 loop using automated and often real-time mining to make frequent modifications to the monitored patterns.

The set of active patterns is usually compiled into a single composite pattern for efficiency by sharing common state and computations (multi-pattern optimization [10, 11]). Further, we may be interested in reporting trends across patterns, e.g., *how many consecutive patterns were triggered by the same user*, which require merging the patterns into a single automaton.

A closed-loop M3 cycle introduces the new challenge of supporting *dynamic patterns*, i.e., patterns that change over time — either individual pattern transitions or the active set of patterns itself. In Example 1, the set of chart patterns or value k may change rapidly over time. In an online advertising system that targets user behavior [9], patterns can vary rapidly when automated mining schemes are used to define currently "hot" behavior rules. Network monitoring rules (see Appendix A) may also change rapidly with time.

The obvious technique for handling dynamic patterns — terminate the old CQ and create a new one — has several disadvantages:

- We lose the in-memory state built up within the operator from events matching the unchanged portion of the pattern.
- We are unable to allow the matching of older events against the new transitions, unless we save and replay past events.
- In the common case where the pattern operator is part of a larger CQ with other stateful operators and possibly multiple outputs, tearing down the entire query to modify the query may be unacceptable. But, replacing a pattern operator in a running query is difficult and potentially expensive.

The challenge is to efficiently support dynamic patterns while avoiding the difficulties above.

3) Stream Disorder and Revisions Out-of-order input is common, particularly when data is combined from multiple sources; for instance, when recognizing patterns in a combined feed across stock exchanges. Disorder may also arise from unreliable or high-latency network links, transient router failures, and expensive in-network CQs. Pattern CQs are highly order-sensitive, and can benefit from native support for out-of-order events. The naive technique of buffering events at the input until we can explicitly eliminate disorder (either based on punctuations [26] or using a timeout [5]) can incur high memory usage and latency.

The current constrained augmented NFA-based proposals target ordered streams. While out-of-order schemes have been proposed for simple regular expressions and sequence queries [16, 17], it is challenging to handle disorder with low memory and latency overhead, while supporting an unconstrained augmented model with dynamic patterns. A closely related challenge is efficiently handling *revisions*, where earlier events are modified, e.g., due to error compensation schemes, at the source or in upstream CQ operators.

4) Operator Semantics Clean semantics is a cornerstone of traditional databases and relational algebra, and is necessary for meaningful operator composability, repeatable and well-understood behavior, query debuggability, and cost-based query optimization [27]. DSMS operators that correspond to database operators (e.g., joins) have inherited the clean semantics of those operators; however, existing proposals for stream pattern detection tend to adopt more operational definitions. There is a need for intuitive and declarative semantics — that cleanly handle dynamic patterns, stream disorder, and event revisions — for pattern CQs.

Contributions We make the following contributions in this work:

- We propose a new pattern execution model called an *AFA*, for *augmented finite automaton*. An AFA is an NFA in which each computation is augmented with a *register*, which has a fixed schema and can be manipulated during transitions. AFAs can directly model complex patterns with arbitrary cycles in the transition graph (such as Q_1 and Q_2 in Example 1).
- We define AFA semantics that are independent of operational aspects and physical properties such as disorder and revisions. We propose new semantics and solutions for dynamic patterns, and discuss operator expressiveness (Sections 2 and 3).
- We demonstrate the power of registers and our execution model, by exhibiting AFAs for novel applications such as uncertain streams, stream cleaning, and chart patterns, without changing the underlying model (Section 4, more details in Appendix A).
- We present algorithms to handle out-of-order events and revisions efficiently, allowing fine-grained control over memory usage, latency, and output size, along with efficient state cleanup using punctuations, and support for dynamic patterns (Section 5).
- We evaluate our AFA operator on Microsoft StreamInsight [3], a commercial DSMS. Results show that we are memory-efficient and can achieve rates of more than 200K events/sec — up to 5× better than competing schemes. Further, we handle dynamic patterns orders-of-magnitude more efficiently than simpler solutions such as operator restart (Section 6).

The appendices present novel techniques to efficiently address several other common requirements: handling negative patterns that look for the absence of certain information, ignorable events that do not contribute to matches, and advanced state cleanup (Appendices D, E, and F). Note that language is not the focus of this paper — users can directly write AFA patterns, and pattern languages such as SASE+ [2, 28], Cayuga [10], and Esper [13] can be compiled easily into an AFA. We refer to appropriate related work throughout the paper, with additional details in Appendix G.

2 Streaming AFA Semantics and Formalism

2.1 Preliminaries — Streams and Events

A *stream* is a potentially unbounded sequence $e_0, e_1, \ldots$ of events. An *event* $e_i = \langle p, c\rangle$ is a notification from the outside world (e.g., from a sensor) that consists of two parts: (1) a *payload* $p = \langle p_1, \ldots, p_k\rangle$, which conforms to a pre-defined event schema $\bar{E}$, and (2) a *control parameter* c that provides metadata about the event.

While the exact nature of the control parameter varies across systems [6, 24, 26], two common notions in the context of pattern CQs over streams [2, 17, 28] are: (1) an event generation time, and (2) a time window, which indicates the period of time over which an event can influence output (i.e., contribute to a valid pattern). We capture these by defining $c = \langle \mathsf{LE}, \mathsf{RE}\rangle$, where the time interval $[\mathsf{LE}, \mathsf{RE})$ specifies the period (or *lifetime*) over which the event contributes to output. The left endpoint (LE) of this interval is the application time of event generation, also called the event *timestamp*.

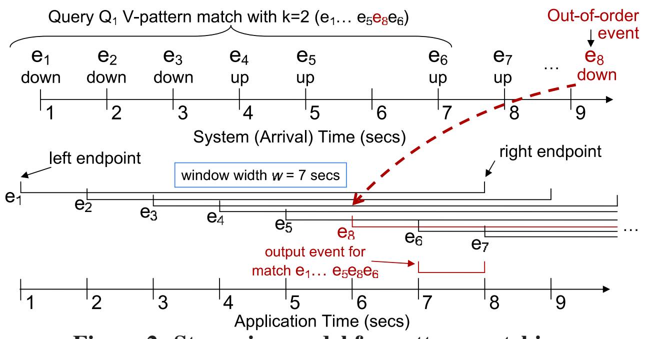


Figure 2: Streaming model for pattern matching.

Assuming a window of width w time units, the right endpoint of an event is simply RE = LE + w. In case there is no windowing, RE is set to ∞. A stream is said to be *disordered* if events may arrive out-of-order with respect to their timestamps.

In order to undo the effect of previously issued events, our streams support revision via *event retraction* [6, 17, 21, 23], where an event serves to remove a previous event from the event sequence. A retraction has the same payload and control parameters as the original event, with an additional bit indicating that it is a retraction.

Figure 2 (top) shows 8 events ($e_1, \ldots e_8$) in arrival order. Here, e_8 is an out-of-order event whose actual timestamp (LE) is 6. Figure 2 (bottom) shows the event lifetimes, assuming a window $w = 7$ secs.

2.2 Streaming AFA Model for Static Patterns

We start with a standard NFA, add the ability to carry additional runtime information with a pre-defined schema, develop new semantics for pattern matching over streams, and apply restrictions to match the streaming setting. Here, we focus on static patterns; formalisms for dynamic patterns are covered in Section 2.3.

AFA Formalism An AFA is a directed graph with labeled nodes called *states*, labeled edges between states called *arcs*, a special *start state*, and a set of special *final states*. In addition, an AFA uses additional computation state, called a *register*, which is associated at runtime with each active state. In order to suit the stream setting, we refine the definition of registers as follows: A register consists of a fixed number of fields $r = \langle r_1, \ldots, r_k\rangle$ and is *schema-based*, i.e., it conforms to a pre-defined register schema $\bar{R}$.

Definition 1 (AFA). *An AFA is a 7-tuple $M = (Q, \mathcal{A}, q_0, \mathcal{F}, Z, \bar{E}, \bar{R})$, where $Q = \{q_0, \ldots q_{n-1}\}$ is the set of states, $\mathcal{A} = \{a_0, \ldots, a_{m-1}\}$ is the set of arcs, q_0 is the start state, $\mathcal{F}$ is the set of final states, Z is the initial register, $\bar{E}$ is the event schema, and $\bar{R}$ is the register schema. Each arc a_i is labeled with a pair of* transition functions *with signatures $f_i(\bar{E}, \bar{R})$ and $g_i(\bar{E}, \bar{R})$, each operating over an event and a register. We refer to these two functions as the* fence function *and the* transfer function *respectively.*

Note that an arc can be defined between any arbitrary pair of states. The fence function $f_i(\bar{E}, \bar{R})$ returns a boolean value that determines if a transition along that arc can be triggered, and, if so, the transfer function $g_i(\bar{E}, \bar{R})$ computes the new register that is associated with the execution after the successful transition. An arc can also be defined as a special ϵ*-arc* that does not consume any event, and whose transition functions operate only over the register.

Example 2 (AFA for Trading). *Figure 3 shows the AFA for the query Q_1 of Example 1. The register consists of a pair of integer fields $\langle r_1, r_2\rangle$. Field r_1 tracks the difference between the number of downticks and the number of upticks across V-patterns, while r_2 tracks the number of consecutive V-patterns detected thus far. We have $Q = \{q_0, \ldots, q_3\}, \mathcal{A} = \{a_0, \ldots, a_5\}, \mathcal{F} = \{q_3\}$, and $Z =$*

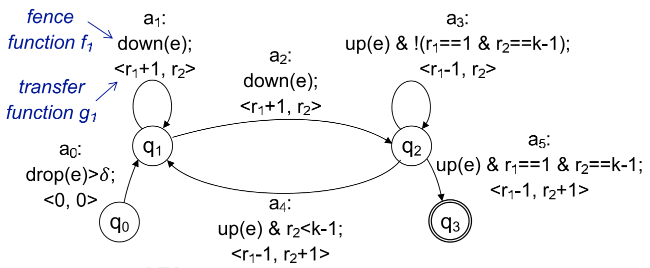

fence function f1
transfer function g1
a1: down(e); <r1+1, r2>
a2: down(e); <r1+1, r2>
a3: up(e) & !(r1==1 & r2==k-1); <r1-1, r2>
a0: drop(e)>δ; <0, 0>
a5: up(e) & r1==1 & r2==k-1; <r1-1, r2+1>
a4: up(e) & r2<k-1; <r1-1, r2+1>
q0
q1
q2
q3

Figure 3: AFA for the V-pattern query Q_1 of Example 1.

$\langle 0, 0\rangle$. Each arc a_i is annotated with fence function f_i (to determine whether the transition is triggered) and transfer function g_i (for the new register content). Methods up(e) *and* down(e) *determine if event e is an uptick or a downtick, while* drop(e) *indicates the magnitude of the drop. For instance, arc a_1 checks if e is a downtick; if yes, it increments r_1 while leaving r_2 unchanged.*

AFA Computation Consider a contiguous ordered event subsequence s. We formalize the computation of an AFA M using an *instantaneous description* (*ID*) [15] of M as a 3-tuple (α, q, r), where α is the subsequence of events that have not been processed, $q \in Q$ is the current state, and r (with schema $\bar{R}$) is the current register. The initial ID is (s, q_0, Z). Transitions are denoted using the *next-ID* relation $\vdash_M$. For each event-consuming arc $a_i \in \mathcal{A}$ from state q to state q', we have the relation $(e\alpha, q, r) \vdash_M (\alpha, q', r')$ if and only if $f_i(e, r)$ is true and $g_i(e, r) = r'$. If a_i is an ϵ-arc, we have the relation $(\alpha, q, r) \vdash_M (\alpha, q', r')$ if and only if $f_i(-, r)$ is true and $g_i(-, r) = r'$.

The relation $\vdash_M$ computes one step of M, while the reflexive, transitive closure $\vdash_M^*$ computes zero or more steps of M. The AFA M is said to *accept* the subsequence s (i.e., recognize the pattern) if $(s, q_0, Z) \vdash_M^* (\varnothing, q, z)$, where $q \in \mathcal{F}$ and $\varnothing$ is an empty sequence.

Example 3 (AFA Computation). *Continuing Example 1 with Q_1, each event is either an uptick or a downtick. We compute the AFA of Figure 3 (with $k = 2$), over the ordered event sequence $s = e_1 \ldots e_5e_8e_6$ of Figure 2. The initial ID is $(s, q_0, \langle 0, 0\rangle)$. When event e_1 is consumed, it triggers the function f_0 associated with a_0, the only outgoing arc from q_0. Assuming e_1 is a price drop greater than δ, the new register is computed using g_0 and the AFA completes the step $(e_1 \ldots e_5e_8e_6, q_0, \langle 0, 0\rangle) \vdash_M (e_2 \ldots e_5e_8e_6, q_1, \langle 0, 0\rangle)$. Next, downtick e_2 can be consumed by both arcs a_1 and a_2, giving two possible next-IDs, $(e_3e_4e_5e_8e_6, q_1, \langle 1, 0\rangle)$ and $(e_3e_4e_5\, e_8e_6, q_2, \langle 1, 0\rangle)$.*

We see that there can be multiple next-ID relations, not all of which lead to acceptance. In our example, the sequence of computations $(e_1 \ldots e_5e_8e_6, q_0, \langle 0, 0\rangle) \vdash_M (e_2 \ldots e_5e_8e_6, q_1, \langle 0, 0\rangle) \vdash_M (e_3e_4e_5e_8e_6, q_1, \langle 1, 0\rangle) \vdash_M (e_4e_5e_8e_6, q_2, \langle 2, 0\rangle) \vdash_M (e_5e_8e_6, q_2, \langle 1, 0\rangle) \vdash_M (e_8e_6, q_1, \langle 0, 1\rangle) \vdash_M (e_6, q_2, \langle 1, 1\rangle) \vdash_M (\varnothing, q_3, \langle 0, 2\rangle)$ leads to s being accepted by M since $q_3 \in \mathcal{F}$.

Streaming AFA Operator We now define the semantics of our streaming AFA operator. The pattern output is described in an order-independent manner by specifying the output stream as a set of events computed in terms of the set of all input events[1].

Definition 2 (Streaming AFA Operator). *Given (1) an AFA $M = (Q, \mathcal{A}, q_0, \mathcal{F}, Z, \bar{E}, \bar{R})$, and (2) an input stream that consists of a (possibly countably infinite) set of events $\mathcal{I}$, let I denote the ordered sequence based on LE, of all events in $\mathcal{I}$, and S denote the set of all possible contiguous subsequences of I. The output stream of a streaming AFA operator O_M is a (possibly countably infinite) set*

[1]As in previous work, we assume that timestamps are unique. A clean extension to non-unique timestamps is straightforward (based on the concept of "multi-events" introduced in Sec. 4). Our algorithms also extend naturally, but the details are omitted for brevity.

of events O defined as follows. An output event $e_i = \langle p_i, c_i \rangle$ where $p_i = z$ and $c_i = \langle \mathsf{LE}, \mathsf{RE} \rangle$, belongs to O iff, for some sequence $s \in \mathcal{S}$, we have $(s, q_0, Z) \vdash_M^* (\varnothing, q, z)$ where $q \in \mathcal{F}$ (i.e., M accepts s) and $[\mathsf{LE}, \mathsf{RE})$ is the intersection of all event lifetimes in s.

In our running example, the event sequence $s = e_1 \dots e_5 e_8 e_6$ forms a valid match for Q_1, resulting in a single output event with the lifetime shown in Figure 2 (bottom). The definition of output lifetime naturally prevents matches across events that are not within the window w (e.g., across e_1 and e_7 in Figure 2).

The events that contribute to a match e can be reconstructed if needed, by looking for events in the input stream whose lifetime $[\mathsf{LE}, \mathsf{RE}) \supseteq [e.\mathsf{LE}, e.\mathsf{RE})$. As an aside, note that associated with any application time t, there is a collection $C(t)$ of input events (ordered by LE) that are *live* at t, i.e., have a lifetime that is stabbed by t. Our operator has the desirable *changing relation* property [18] that an output event is live at t if and only if $C(t)$ contains the corresponding event subsequence that M accepts.

2.3 Adding Support for Dynamic Patterns

The goal is to seamlessly handle dynamic patterns, i.e., patterns where arcs (and the associated states) may get added or removed with time. The key idea behind our solution is to treat the AFA arcs $\mathcal{A}$ as a second streaming event input to the operator.

An *arc-event* e_a for an arc a from state q_x to state q_y is an event that contains a payload and a lifetime. The payload has the form $\langle q_x, q_y, f, g, isFinal \rangle$. Here, *isFinal* is a boolean that, in case q_y is a newly added state to the AFA, indicates whether $q_y \in \mathcal{F}$. Functions $f(\bar{E}, \bar{R})$ and $g(\bar{E}, \bar{R})$ are as defined earlier. Deletion of existing arcs is performed using arc-event retraction.

The arc-event e_a has a lifetime $[e_a.\mathsf{LE}, e_a.\mathsf{RE})$. The semantics of AFA computation are modified to take arc lifetimes into account. Specifically, the next-ID relation for an arc-event e_a corresponding to event-consuming arc a is $(e\alpha, q, r) \vdash_M (\alpha, q', r')$ iff $f(e, r)$ is true, $g(e, r) = r'$, and $e_a.\mathsf{LE} \le e.\mathsf{LE} < e_a.\mathsf{RE}$. If a is an ϵ-arc, we have $(\alpha, q, r) \vdash_M (\alpha, q', r')$ iff $f(-, r)$ is true, $g(-, r) = r'$, and $e_a.\mathsf{LE} \le e.\mathsf{LE} < e_a.\mathsf{RE}$, where e is the event whose consumption (indirectly) triggered e_a. In other words, given an arc-event e_a with lifetime $[e_a.\mathsf{LE}, e_a.\mathsf{RE})$, only events with a timestamp stabbing $[e_a.\mathsf{LE}, e_a.\mathsf{RE})$ can trigger arc a.

Note that we do not choose *temporal-join-style* semantics for dynamic patterns, where an event triggers an arc-event if their lifetimes intersect. The reason is that these semantics are less intuitive for patterns — an arc-event can affect an event with an earlier timestamp, even if the arc did not exist at event occurrence time. Further, these semantics lead to a severe inefficiency in state cleanup; we cover the details in Section 5.4.

Practical Lifetime Restrictions The most common usage scenario for dynamic patterns is the case where users want an arc insertion (or deletion) to apply to all future events from the point of insertion forwards. We support this *default operation mode*, where users do not explicitly specify arc-event lifetimes. Let $t_{curr} = \max(t_e, t)$ where t denotes the largest timestamp across all events received on the first input to the operator. When a new arc-event e_a is received on the second input, its lifetime is implicitly set to (t_{curr}, ∞). Arc deletions correspond to a change in arc-event lifetime from the old lifetime (LE, ∞) to the new lifetime $(\mathsf{LE}, t_{curr}]$.

Example 4 (Dynamic Patterns)**.** *Refer to the running example in Figure 3. Each arc $a_0, \dots, a_5$ is associated with a lifetime of $[-\infty, \infty)$. The change of k to k′ is expressed by first deleting arcs a_3, a_4, and a_5, i.e., changing their lifetime to $[-\infty, t_{curr}]$, and then inserting three arcs with lifetimes (t_{curr}, ∞) that use the new value k′. Future events will produce output only if k′ V-patterns are encountered.*

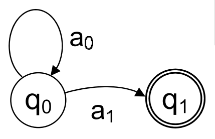


Figure 4: Two-state AFA.

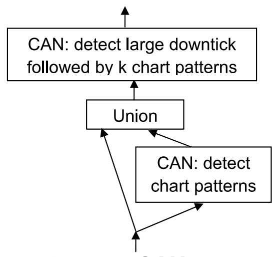

CAN: detect large downtick followed by k chart patterns
Union
CAN: detect chart patterns

Figure 5: CAN plan.

2.4 Speculation and Punctuations

Speculation We defined the streaming AFA operator's semantics declaratively, in the presence of disorder. Our implementation operationally ensures that on any prefix of the input, the output event stream adheres to the semantics above. Thus, we may produce output that may need to be withdrawn subsequently due to an out-of-order input event — we refer to this as *speculation*. Incorrect output events are withdrawn by issuing retraction events. We can handle speculative input and produce maximally speculative output, but aggressive speculation may not always be desired; Section 5.3 describes techniques for controlling speculation.

Punctuations We need a way to ensure that an event is not arbitrarily out-of-order. Otherwise, we can never declare any produced output to be "final", i.e., it cannot change due to future events. This facility is useful when we need to prevent false-positives in scenarios where correctness is important [17], such as directing an automatic power plant shutdown based on detected anomalies. Further, we cannot clean historic state in the DSMS, since it may be needed forever in order to adjust previous output.

Our notion of stream progress is realized using time-based *punctuations* [6, 24, 26]. A time-based punctuation is associated with a timestamp t and indicates that there will be no future event with timestamp $< t$. Section 5.4 shows how punctuations can provide output guarantees and perform state cleanup for AFAs. Appendix F introduces predicate-based punctuations to further optimize cleanup.

3 AFA Expressiveness

In this section, we compare AFAs (with static patterns) to constrained augmented NFAs (CAN) in terms of expressiveness. Further details on AFA expressiveness, including a comparison to classical automata classes, can be found in Appendix B.

A CAN-based model (see Section 1) cannot express patterns with arbitrary arcs, such as the one in Figure 3. It might appear that this limitation of CAN can be overcome as follows. First, add register entries to maintain the set of "active" NFA states (thus, the register uses $O(n)$ space, where n is the number of NFA states). Next, define a simple two-state AFA (see Figure 4), where the self-loop transition a_0 encodes all actions corresponding to the original automaton, updating register contents to simulate transitions to subsequent NFA states. However, this strategy has problems:

- It is less efficient because of the overhead of embedding the automaton within a transition. We will see in Section 6 that the overhead is up to 25% for our workloads.
- It amounts to writing a custom function for a specific pattern, as opposed to using a general operator that handles any automaton.
- The self-loop just feeds events sequentially to the user-code, and thus we can no longer perform operator-level optimizations such as efficient handling of disorder, ignorable events, etc.
- Dynamic patterns cannot be supported directly using this scheme.
- Supporting arbitrary NFA graphs allows one to understand what regular language the pattern is a subset of, seems a more natural

way of expressing the pattern, is easier to modify, and makes implementation cleaner when there are multiple partial matches "in flight" at once.

Another alternative is to partition the pattern into multiple CAN operators in a CQ plan. For example, an AFA that looks for a large price drop followed by k chart patterns (where k may be data-dependent) can be created using the query plan in Figure 5, where one CAN detects every chart pattern, and another CAN detects a large price-drop followed by k chart patterns. However, this solution is highly inefficient: (1) we need to detect every chart pattern even though we are interested in chart patterns only after a large price drop, and (2) there is overhead due to several operators and event queues in the plan. Further, this alternative makes writing the pattern CQ harder as compared to directly specifying an automaton.

4 Practical Examples of Using AFAs

We illustrate the generality of our approach by casting a variety of applications into the AFA model. We next discuss uncertain streams; in Appendix A, we discuss how the AFA can be specified by a user or compiled, along with more examples such as stream data cleaning, complex chart patterns, and network monitoring.

Uncertain Streams Uncertain data streams, where the content of each event is not known with certainty, are becoming increasingly commonplace [12], for example, in RFID/GPS networks and environmental monitoring. Assume that each event e_i has a probability p_i of being present in the stream, and that p_i is stored as a column (say prob) in the event schema. For example, if the readings of a particular RFID reader are spurious 10% of the time, each event would have prob $= 0.9$. We want each pattern CQ output event to be associated with a probability that the pattern actually occurred.

We modify an AFA that matches a desired pattern over a traditional (certain) stream as follows: we add an entry (r_{prob}) with default value 1 in the register to track the probability of pattern occurrence. Each successful arc transition due to an event e_i updates the register value to $r_{prob} \times p_i$. In addition, we add a self-loop transition that remains in the same state and sets the new register value to $r_{prob} \times (1 - p_i)$, to model the non-occurrence of e_i. This solution can lead to a proliferation of partial matches, and is controlled by setting some output probability threshold below which further matching is discontinued. Note that we were able to support uncertain streams without modifying the DSMS or the AFA model.

We can also let each event take on different values with varying probabilities. For example, a sensor may produce events reporting an object's color as blue with probability $p_{blue} = 0.7$, indigo with probability $p_{indigo} = 0.2$, and green with probability $p_{green} = 0.1$. Here, we model the alternative values as a "multi-event" that contains value-probability pairs. Any use of e.color $= a$ in a fence function becomes $p_a > 0$, and the corresponding transfer function is used to update a cumulative probability in a register: $r_{prob} = r_{prob} \times p_a$. This construction does not change the number of states and arcs in the AFA, and the uncertainty is handled by the existing mechanisms for managing multiple in-flight partial matches.

5 Implementing the AFA Operator

We present the basic algorithm for static patterns with disordered streams in Section 5.1, while Section 5.2 covers dynamic patterns. Speculation control and cleanup are covered in Sections 5.3 and 5.4, while further optimizations are covered in the appendices.

5.1 Algorithms for Static Patterns

Storing Arcs and Registers Our operator maintains an *arctable* — a hash table indexed by state. For each state q, arctable contains

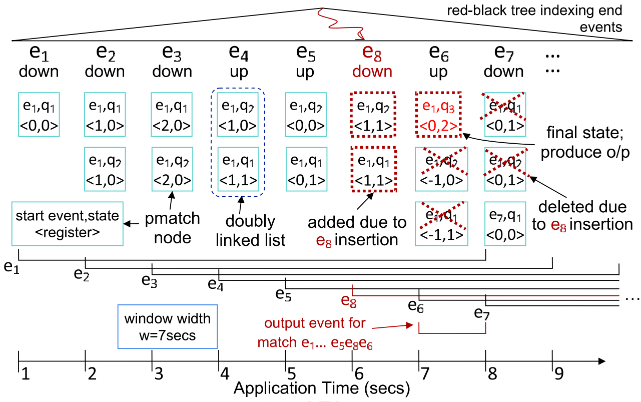


Figure 6: Data structure for AFA operator, showing the insert of out-of-order event e_8. Event lifetimes are also shown.

a list of arcs that originate from q. Each arc is a structure with pointers to the fence and transfer functions (f_i and g_i). Since registers are similar to events (with a predefined schema), we leverage the event infrastructure to store registers in a common in-memory page pool. Thus, registers inherit the benefits of events, such as null suppression. We only manage pointers to events and registers; hence, the terms "events" and "registers" below refer to these pointers.

Data Structures We use a structure called *pmatch* (for *partial match*) to store computation state for potentially incomplete pattern matches. A pmatch is conceptually associated with an ordered event subsequence $e_0 \ldots e_k$, where e_0 and e_k are referred to as the StartEvent and EndEvent of the pmatch (their lifetimes are denoted by [StartLE, StartRE) and [EndLE, EndRE) respectively). A pmatch node uses four fields (described below) to concisely represent an ID that is derivable from the initial ID by executing the AFA on the subsequence $e_0 \ldots e_k$. Note that the corresponding output match lifetime is [EndLE, StartRE).

The pmatch nodes are organized in an efficient data structure called rbtree, which uses a red-black tree (see Figure 6) to index each event by its timestamp (LE). For every event e indexed in the tree, rbtree maintains a doubly linked list of all pmatch nodes with EndEvent e. We maintain the following invariants for rbtree:

- (Content) In the absence of punctuations, for every in-order subsequence $e_0 \ldots e_k$ of received events where e_0.RE $> e_k$.LE, there exists a pmatch node corresponding to every ID $(\varnothing, q, r)$ s.t. $(e_0 \ldots e_k, q_0, Z) \vdash_M^* (\varnothing, q, r)$ where $q \in Q$. Each pmatch node contains 4 fields: StartLE $= e_0$.LE, StartRE $= e_0$.RE, q, and r. These pmatch nodes are stored in the linked list for e_k in rbtree (thus, EndLE and EndRE are available to the pmatch node).
- (Ordering) The pmatch entries in any linked list are maintained in increasing order of StartLE. We will see shortly that we can maintain this ordering efficiently.
- (Completeness) The presence of a pmatch entry for the event sequence $s = e_0, \ldots, e_k$ implies the existence of k other pmatch entries in the tree (one for each distinct prefix of sequence s).

Insert Algorithm We now give a short description of the insertion algorithm. More details, along with runtime complexity, and the deletion algorithm, are covered in Appendix C. Algorithm 1 shows the insertion algorithm (we omit the handling of ϵ-arcs for simplicity). The method Search$_{\leq}(t)$ returns the entry in rbtree with timestamp closest to and $\leq t$. The method Search$_{\geq}(t)$ is also defined similarly. Both are $O(\lg r)$ operations, for r events in rbtree.

If the inserted event e is out-of-order (with a timestamp t), we first call RemoveInvalidatedSequences (Lines 10—19) to delete partial matches whose subsequences span across t and hence are no

Algorithm 1: AFA insertion (handling out-of-order input).

```
1  InsertEvent(event $e$) begin
2      if out of order then RemoveInvalidatedSequences($e$.LE);
3      entry $x \leftarrow$ rbtree.Search$_\leq$($e$.LE) ;      // Get previous pmatch list
4      entry $y \leftarrow$ new entry; $y$.event $\leftarrow e$; $y$.list $\leftarrow$ MakeTransitions($x$.list, $e$);
5      $y$.list.Append(NewTransition($q_0, Z, e$) ; // New pmatch from state $q_0$
6      foreach pmatch $p$ in $y$.list do  if $p.q \in \mathcal{F}$ then $p$.InsertOutputEvent();
7      rbtree.AddEntry($e$.LE, $y$) ;     // List $y$ will be in StartLE order
8      if out of order then PropagateInsert($y$);
9  end
10 RemoveInvalidatedSequences(timestamp $t$) begin
11     entry $e \leftarrow$ rbtree.Search$_\geq$($t$); boolean done $\leftarrow$ false;
12     while not done do
13         done $\leftarrow$ true ;                    // Early stopping condition
14         pmatch $p \leftarrow e$.list.Head;
15         while $p$.StartLE $\leq t$ do           // Delete invalid pmatches
16             if $p.q \in \mathcal{F}$ then $p$.RetractOutputEvent();
17             done $\leftarrow$ false; delete $p$; $p \leftarrow e$.list.Head;
18         $e \leftarrow e$.Next;
19 end
20 PropagateInsert(entry $y$) begin
21     list $z \leftarrow y$.list;
22     while $z$ is not empty do
23         $y \leftarrow y$.Next; $z \leftarrow$ MakeTransitions($z$, $y$.event);
           // Reverse append to maintain (Ordering) invariant
24         $y$.list.AppendFront($z$.Reverse)
25 end
```

longer valid. If a deleted pmatch corresponds to a final state, we output a retraction event to compensate for the invalid prior output. Invariant (Completeness) allows us to ensure that the algorithm visits no more pmatch nodes than the minimum necessary.

We next (Lines 3—7) retrieve the pmatch list corresponding to the immediately previous event, and apply transitions (using arctable) to each outgoing arc for each pmatch. We also attempt to start a new match (from q_0) beginning at event e. If any new pmatch state is final, we produce an output event. Event e is added to rbtree and associated with the new list of pmatch nodes, which follows the (Ordering) invariant by construction. Finally, if e is out-of-order, we continue the matching process with subsequent event entries in rbtree (Lines 20—25) until no new pmatch nodes get created. During this process, we can maintain the (Ordering) invariant without having to sort the pmatch lists (see Line 24).

Runtime complexity is covered in Appendix C; briefly, we pay a higher price only for out-of-order events, with lower overhead if an event is only slightly out-of-order. We finally note that our algorithms access events and registers in a column-major order (in rbtree), which makes them more cache-friendly than using horizontal pointers between related pmatch nodes.

Example 5 (AFA Insertion)**.** *Figure 6 shows the data structure for our running example, after $e_1 \ldots e_5 e_6$ have arrived and e_8 is being inserted. We delete four invalidated* pmatch *nodes as shown, and then create* 2 *new nodes for e_8. One of these matches is propagated to entry e_6 as a new* pmatch *— this is at the final state and causes output of an event with the lifetime shown.*

5.2 Algorithm Changes for Dynamic Patterns

Arc-events are stored in arctable as before, along with their lifetimes. Before applying an AFA computation, we ensure that the arc is valid for the computation based on the semantics defined in Section 2.3, i.e., the event timestamp is contained within the corresponding arc-event lifetime.

Algorithms Under the default operation mode (cf. Section 2.3), we do not need to do anything extra when there is an arc-event insertion or deletion, other than updating arctable. This is because under this mode, arcs do not affect existing events. Our semantics ensure that existing pre-computed state (partial matches) will remain valid for the new arcs and future events.

On the other hand, when arc-events are associated with explicit user-specified lifetimes, on the insertion of an arc-event e_a with lifetime $[e_a.\mathsf{LE}, e_a.\mathsf{RE})$ from state q_x to q_y, we invoke the method Search$_\geq$(e_a.LE) to locate the first affected event in rbtree. We traverse the linked list associated with the previous event to locate partial matches ending at q_x, and apply the new transition to each of them (if q_x is the start state, new matches are also started as before). If q_y is a final state, we may generate new output matches. This is repeated for each event whose LE stabs the lifetime of arc-event e_a. Note that any new pmatch entries created during this process also need to be matched with further events, similar to the PropagateInsert procedure in Algorithm 1. Arc-event deletion is similar but slightly more complicated; it is omitted for brevity.

5.3 Controlling Operator Speculation

The algorithms above are maximally speculative; i.e., when there are events with consecutive timestamps, we output matches across them. Thus, an out-of-order event between them may cause retractions. We have two techniques for limiting speculation.

Leveraging Event-Ordering Information In many applications (e.g., sensors and stocks), it may be possible for the source to provide additional ordering information in the form of a *sequence number* that increases by 1 for every event. We can leverage sequence numbers to build an optimized AFA operator called *AFA+O*.

1. We do not perform AFA transitions across events with sequence numbers that are not adjacent. In this case, the algorithm is speculation-free in terms of output, and at the same time performs maximal precomputation for expected out-of-order events.
2. We index events by sequence numbers instead of timestamps. We can replace the red-black tree with a hash table on sequence numbers, since the calls Search$_\leq$(x) and Search$_\geq$(x) can now be replaced by hash table lookups for x and $x - 1$ or $x + 1$.

Controlling Speculation with Cleanse We feed the input stream into an operator called *Cleanse* that is placed before the AFA. Cleanse accepts a *speculation factor* σ as parameter. If the latest punctuation has timestamp t, Cleanse maintains the invariant that only events with a timestamp less than $t + \sigma$ are propagated. Other events are buffered in-order within Cleanse. Thus, when a new punctuation with timestamp t' arrives, Cleanse releases the buffered events with timestamp less than $t' + \sigma$, in timestamp order. We can control speculation by varying σ. For example, $\sigma = 0$ implies that the Cleanse output is always in-order, and forces AFA into zero speculation. Similarly, $\sigma = \infty$ causes AFA to be maximally speculative. We can also instead place Cleanse at the output of AFA to control only the output size (details omitted for brevity).

5.4 State Cleanup using Punctuations

Let t_e and t_a denote the latest punctuations along the event and arc inputs respectively. For the purpose of AFA state cleanup, the *effective* incoming punctuation for the AFA operator is $t_p = \min(t_e, t_a)$. We cover cleanup in detail in Appendix C.1; briefly, whenever t_p increases, we perform state cleanup and output a punctuation t_p. Cleanup is based on the following insight: since there can be no new event with a timestamp before t_p, the earliest possible out-of-order event (or arc) will require the presence of no event earlier than the latest event with timestamp $< t_p$. Further, an arc-event e_a can be discarded when $t_p \geq e_a.\mathsf{RE}$.

Our dynamic pattern semantics allow us to cleanup an event e when the punctuation crosses e.LE. The alternate temporal-join-style semantics, where an event is affected by an arc-event if their lifetimes intersect, would imply that we need to retain an event e

parameter	value
number of events	100K
probability of uptick p	$0.5, 0.7$
initial price drop δ	\$0.25
price change	$N(0.70, 0.30)$
window size W	500–2000
%ignorable events I	0–90%
number of V-patterns k	1–3
pattern CQs	Q_1, Q_2, Q_3

Figure 7: Experimental parameters.

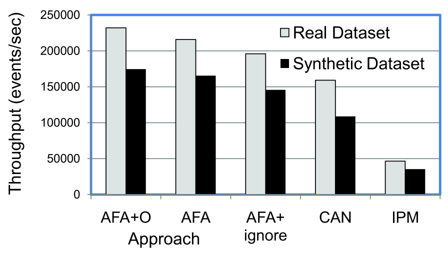


Figure 8: Throughput across approaches.

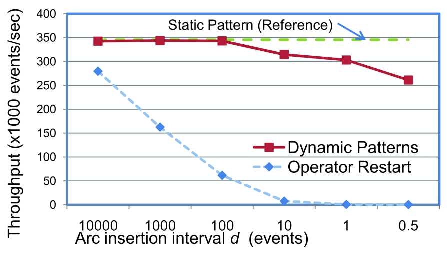


Figure 9: Throughput; dynamic patterns.

until the punctuation crosses e.RE; before this time, an inserted arc-event could intersect e's lifetime and require AFA computation with e. This can be very inefficient for large window sizes.

6 Evaluation

We use Microsoft StreamInsight [3], a commercial DSMS, to run all experiments. The experiments are performed on a 3.0GHz Intel Core 2 Duo PC with 3GB main memory, running Windows Vista. Our AFA operator (around 1000 lines of C# code) supports speculation, disorder, dynamic patterns, and the optimizations proposed in this paper. We evaluate several flavors of our AFA operator:

- AFA: The basic operator with dynamic patterns (cf. Section 5).
- AFA+O: AFA that leverages sequence numbers (cf. Section 5.3).
- AFA+ignore: An enhanced AFA operator that optimizes for *ignorable arcs*, to efficiently ignore intermediate events that are not part of the pattern (cf. Appendix D for details).

In order to compare against dynamic patterns, we implement *Operator Restart* as an alternative: whenever the pattern changes, we replace the old AFA operator in the CQ plan with a new AFA that uses the new pattern specification, and feed it the currently active events (from a buffer) so that no matches with existing events are lost. We also implemented the Cleanse operator (see Section 5.3) with a user-specified speculation factor σ to order/release events.

6.1 Metrics and Workloads

We track: (1) *Throughput*, which measures the number of events processed per second. (2) *Latency*, which indicates the difference between the system time of a pattern output and the arrival time of the latest event contributing to the output. (3) *Memory*, which measures the amount of memory used for storing registers and pmatch entries in the operator. (4) *Output Size*, which is the number of output events produced, including retractions (i.e., chattiness).

Synthetic Dataset We use a stock trading application, where each event is a stock quote with fields symbol, price, and price change. We generate 100K events using an event generator that models the workloads used in Agrawal et al. [2]. Price has a probability p of rising (uptick) and $1 - p$ of falling (downtick), with magnitude drawn from a truncated normal distribution with mean \$0.70 and standard deviation \$0.30. This is the default dataset in experiments.

Real Dataset We approximate a transactional stock ticker by collecting historical opening prices (mined from Yahoo! Finance) of one stock (YHOO) from years 1996 to 2009, to generate an event trace. This dataset is extrapolated into a larger dataset of 100K events by repeating the observed price variations multiple times.

Queries We test 3 queries. Queries Q_1 and Q_2 are shown in Example 1 (δ is the price drop). Note that k is data-dependent in Q_2. Query Q_3 is a register-free query looks for a linear sequence of 8 ticks. Each arc in Q_3 is randomly looks for an uptick or a downtick.

We use a sliding window with W events per window. There is one punctuation every W events. For some experiments, I% of events are converted into random ignorable events that are irrelevant to the pattern match. Events are preloaded from logs into memory, to avoid I/O bottleneck. Default parameters are summarized in Figure 7. We report additional results in Appendix D.

6.2 Comparing AFAs to Other Techniques

We compare the AFA operator against two alternative proposals:

- IPM (*Iterative Pattern Matcher*): This is an optimized pattern CQ implemented using standard stream operators, based on the recursive stream query processing strategy proposed recently [7]. Briefly, IPM consists of two join operators that join the streaming events with AFA transitions and with earlier partial matches. New pattern matches are fed back to the second join operator to enable subsequent matches. IPM handles out-of-order events, but requires explicit source-provided sequence numbers.
- CAN (*Constrained Augmented NFA*): We use a two-state CAN that embeds the original unconstrained automaton (cf. Section 3) to achieve AFA's level of expressiveness. This scheme uses our AFA+ignore implementation, thus deriving all the benefits of our algorithms. The main purpose of comparing to CAN is to evaluate whether we pay a performance penalty for natively supporting queries with arbitrary arcs. Note that CAN corresponds to the execution model used in recent research [2, 10] (assuming static patterns and ordered input).

We use query Q_1 ($k = 2$) with both the real dataset ($\delta = 0.05$) and the synthetic dataset ($p = 0.5, \delta = 0.25$), and compare throughputs when processing in-order data with no ignorable events. The window size is $W = 2000$ events. Figure 8 shows that we get the highest throughput when we leverage sequence numbers (AFA+O). The basic AFA operator is slightly worse, followed by AFA+ignore. CAN has lower throughput[2] than AFA+ignore (around 16% for real data and 25% for synthetic data), due to the overhead of embedding the original AFA within the outer CAN.

We do not evaluate the query-plan-based alternative described in Section 3, since it can be made arbitrarily worse than other techniques (by increasing δ). Finally, IPM does 5× worse than our operators, for the same input data and pattern CQ. This difference is because IPM does not use our specialized algorithms to optimize for pattern matching. (The added overhead of enqueuing and dequeuing events across multiple operators was found to be marginal.)

6.3 Detailed Evaluation of AFAs

1) Evaluating Dynamic Patterns We set $W = 500$ and $p = 0.5$ so that upticks and downticks are equally probable. We use Q_3, a linear AFA with 8 arcs, with the synthetic in-order dataset. Every d events, we delete a random arc from the AFA and insert a new arc in its place, that converts an uptick into a downtick or vice versa. This change does not affect the probability of a match; thus, any variations in throughput are only due to the overhead of dynamic patterns and not due to the new pattern itself. Figure 9 shows throughput as we vary the arc insertion interval d. Throughput (~ 350K

[2]We also found that CAN uses around 8% more memory than AFA+ignore, primarily due to the larger register.

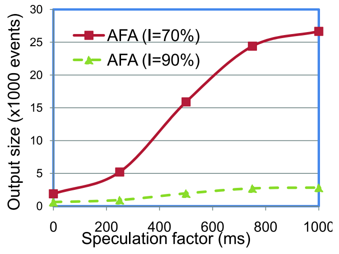


Figure 10: Output size vs. speculation.

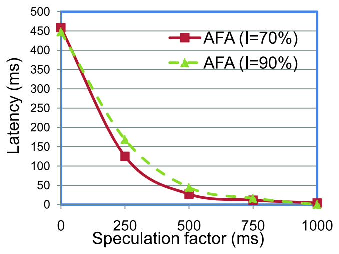


Figure 11: Latency vs. speculation.

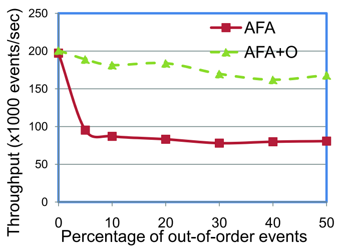


Figure 12: Throughput vs. disorder.

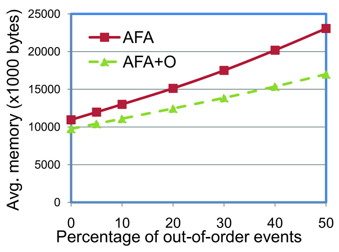


Figure 13: Memory utilization vs. disorder.

events/sec) for a static pattern is also shown. We see that even in the case of an arc change every 10 events, the overhead incurred by dynamic patterns is less than 10%. In contrast, *Operator Restart* degenerates quickly due to the overhead of replacing the AFA and rebuilding state whenever the pattern changes. Dynamic patterns were also found to have negligible impact on memory (not shown).

2) Controlling Speculation We use Q_1 with the synthetic dataset ($k = 1$, $p = 0.5$, $W = 500$, $\delta = 0.25$). 5% of events are out-of-order by between 1–5000 events. We use the optimized AFA+ignore operator, and set the ignorable event percentage (I) to 70% and 90%. One event arrives at the DSMS every millisecond. The Cleanse operator is used to control aggressiveness. As we increase the speculation factor σ from 0 to 1000ms, Figure 10 shows that there is more aggressive speculation leading to greater output size (in terms of number of events). Moreover, $I = 90\%$ causes lesser relative increase (4.5×) in output size compared to $I = 70\%$ (14×) with increasing speculation — this is because AFA+ignore ensures that ignorable out-of-order events do not cause the operator to unnecessarily retract and reissue matches. With AFA+O, output size is always the same as that for AFA when $\sigma = 0$ because we can leverage sequence numbers to output events only when certain, *without* having to wait for punctuations. Also, latency (see Figure 11) decreases with σ as expected. Maximal speculation ($\sigma = \infty$) was found to reduce the latency to less than 0.5ms.

3) Increasing Out-of-order Events We use CQ Q_2 with $p = 0.7$ and $W = 500$, and increase disorder by moving events in the stream. We shift ω% of events randomly by between 1–1000 events. The Cleanse operator is not used. We compare AFA and AFA+O. Figure 12 shows throughput as we increase ω. AFA and AFA+O are comparable when there is no disorder, giving an event rate of ~ 200K events/sec. Disorder reduces throughput in the absence of sequence numbers, since speculative matches may need to be corrected. AFA+O gives the highest throughput and is mildly affected by disorder. Figure 13 shows memory consumption; we see that it increases with disorder as expected, with AFA+O degrading slower because it avoids unnecessary matches due to out-of-order events.

7 Conclusions

Current models for pattern CQs over streams considerably improve expressiveness over regular expressions. We continue in this direction, and further enhance the model to better handle new applications — our AFA model allows unrestricted transitions and supports out-of-order input and revisions, while maintaining or improving efficiency. We further motivate and solve the new problem of efficiently handling *dynamic patterns*, which are important for modern business applications. To our knowledge, this is the first attempt at defining declarative order-agnostic stream semantics for the AFA model with dynamic patterns and augmented with schema-based registers. AFAs are versatile and can be used in innovative ways, e.g., over uncertain streams. We describe novel algorithms to implement the AFA operator, and present new optimizations. Experiments on Microsoft StreamInsight show that we achieve rates of more than 200K events/sec (up to 5× better than simpler schemes). Dynamic patterns can deliver orders-of-magnitude better throughput than solutions such as operator restart, and our optimizations are effective at improving or controlling several other metrics.

References

[1] D. Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.
[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.
[3] M. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. In *VLDB*, 2009.
[4] B. Babcock et al. Models and issues in data stream systems. In *PODS*, 2002.
[5] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM TODS*, 2004.
[6] R. Barga et al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
[7] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. In *VLDB*, 2009.
[8] Chart Patterns. `http://tinyurl.com/6zvzk5`.
[9] Y. Chen et al. Large-scale behavioral targeting. In *KDD*, 2009.
[10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.
[11] Y. Diao et al. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 2003.
[12] Y. Diao et al. Capturing data uncertainty in high-volume stream processing. In *CIDR*, 2009.
[13] EsperTech. `http://esper.codehaus.org/`.
[14] M. Franklin et al. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.
[15] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
[16] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *ICDE*, 2007.
[17] M. Liu et al. Sequence pattern query processing over out-of-order event streams. In *ICDE*, 2009.
[18] D. Maier et al. Semantics of data streams and operators. In *International Conference on Database Theory*, 2005.
[19] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *SIGMOD*, 2008.
[20] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, 2009.
[21] R. Motwani et al. Query processing, approximation, and resource management in a DSMS. In *CIDR*, 2003.
[22] Oracle Inc. `http://www.oracle.com/`.
[23] E. Ryvkina et al. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.
[24] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.
[25] StreamBase Inc. `http://www.streambase.com/`.
[26] P. Tucker et al. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 2003.
[27] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.
[28] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

APPENDIX

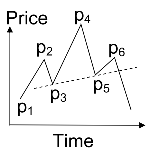
A AFA Specification and Examples

We focus on the specification provided as input to the AFA execution model. Beyond compiling existing pattern languages [10, 13] to AFAs, an alternative, which we find to be convenient and flexible in practice, is a front-end tool that allows users to construct the AFA directly. The AFA operator accepts the AFA specification as a set of states and arcs. For each arc a_i, the specification provides the source and destination state, and the two functions f_i and g_i. The functions are specified in languages such as C++ or C#, or SQL-like expressions that are type-checked, bound to the event and register schemas, and converted into code at CQ compile-time. For instance, the arc a_1 in Figure 3 can be succinctly written using anonymous functions (e.g., .NET *lambda expressions*):

f_1: `(e, r) => down(e)`
g_1: `(e, r) => new Register(r.r1+1, r.r2)`

Stream Data Cleaning Sensor data cleaning [31] is becoming an important application for streams. Streaming data can be cleaned using multiple *stages*, each of which is a CQ to clean the stream in different ways. AFAs can be used to perform some stages of cleaning certain kinds of data. For example, with RFIDs on books, there might be a standard pattern of how a book moves through a library (remove from shelf, check out, re-shelf, etc.). If certain events are missing in a pattern, we can use an AFA to "impute" them. In other words, the AFA recognizes the expected pattern with a missing step, and outputs a "fill-in" event with that step. Note that this process may require complex calculations (e.g., interpolation) based on state accumulated from the other events.

Chart Patterns Consider the more complicated *head and shoulders* [30] chart pattern (on the right), where we look for a trading pattern that starts at price p_1, moves up to local maximum p_2, declines to local minimum $p_3 > p_1$, climbs to local maximum $p_4 > p_2$, declines to local minimum $p_5 > p_1$, climbs again to local maximum $p_6 < p_4$, and finally declines to below the starting price p_1. We can use the AFA of Figure 14 to detect this pattern, where we use three registers (r_1, r_2, r_3) to track prices p_1, p_2, and p_4 respectively.



Network Monitoring Network intrusion detection systems (NIDS) perform deep packet inspection on network streams to detect security threats such as worms and viruses. For example, the Snort NIDS [33] has more than 1000 regular expressions, that are combined into a single automaton for efficiency [19]. As threats become harder to detect, patterns will get more complicated and we may need to maintain information (e.g., IP address) across NFA transitions — suggesting the use of AFAs for network monitoring. Further, the M3 cycle (see Section 1) implies that modern NIDS systems could mine network traces in real-time to add (or update) patterns rapidly in the current set. This requires efficient dynamic patterns to handle pattern churn at high data rates.

B AFA Expressiveness

An AFA is an unconstrained NFA with registers and transitions. In Section 3, we discussed the effect of unconstrained arcs on expressiveness. We now consider the effect of registers and transitions.

Classical automata classes such as Turing machines (TM), linear bounded automata (LBA), push-down automata (PDA), and NFA can be achieved by controlling the register size and transition function power. Functions can either be *arbitrary* or *bounded*, i.e., depend only upon the input event, current state, and a constant part of

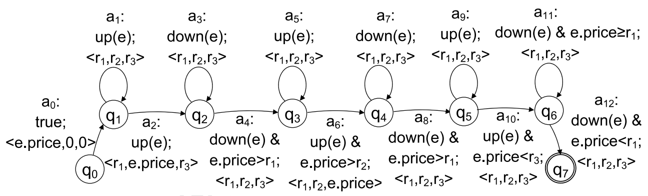

a1: up(e); <r1,r2,r3>
a3: down(e); <r1,r2,r3>
a5: up(e); <r1,r2,r3>
a7: down(e); <r1,r2,r3>
a9: up(e); <r1,r2,r3>
a11: down(e) & e.price≥r1; <r1,r2,r3>
a0: true; <e.price,0,0>
q0
q1
a2: up(e); <r1,e.price,r3>
q2
a4: down(e) & e.price>r1; <r1,r2,r3>
q3
a6: up(e) & e.price>r2; <r1,r2,e.price>
q4
a8: down(e) & e.price>r1; <r1,r2,r3>
q5
a10: up(e) & e.price<r3; <r1,r2,r3>
q6
a12: down(e) & e.price<r1; <r1,r2,r3>
q7

Figure 14: AFA for head & shoulders chart pattern.

the register. Let n be the number of events consumed by the AFA.

- For a TM, we copy the entire input into registers, and then "execute" the TM. With bounded functions, execution requires unbounded registers and ϵ-arcs (to keep processing after input is copied). With arbitrary functions, we just need $O(n)$ registers to hold input — the "execution" is encapsulated in the function.
- An LBA needs only $O(n)$ register state with ϵ-arcs and bounded functions, since it only accesses tape length linear in input size.
- We can express a PDA with $O(n)$ register size (acting as a stack) and bounded functions (that only perform push, pop, and test).
- A single $O(\lg n)$ register (acting as a counter) with bounded (increment and decrement) functions gives us (single-)counter automata — a superclass of regular expressions and a subclass of context-free languages, which can express, for example, $(a^i \cdot b^i)$.
- Multiple $O(\lg n)$ registers with increment and decrement functions can express $(a^i \cdot b^i \cdot c^i)$, which is not a context-free language.
- $O(1)$ fixed-size registers can express regular expressions.

It is also worth noting that AFAs are within the expressiveness of *recursive augmented transition networks* [34] for any given register size and function restriction.

It may appear that $O(1)$ fixed-size registers theoretically do not increase expressiveness beyond regular expressions. However, the reality for pattern specification and execution is different. For example, assume we have a register that can take k possible values (*categories*), e.g., colors. An AFA with $O(N)$ states requires $O(N \cdot k)$ states in the equivalent NFA, with correspondingly more arcs. It quickly becomes unwieldy to express even simple patterns using such a model. The situation is worse with counters, even if bounded — a 32-bit integer register requires 2^{32} states in the equivalent register-free NFA, for every state in the AFA. For example, an AFA with a 32-bit counter and 3 states can accept $(a^i \cdot b^i)$ for $i < 2^{32}$ — this requires around 2^{33} states in the equivalent NFA. This is clearly impractical, thus motivating the need for registers.

In summary, as compared to simpler models, AFAs with fixed registers and unconstrained arcs allow concise and natural expression of many common pattern requirements, and provide greater execution efficiency while avoiding a blowup in the number of automaton states or a need for more complex query composition.

C AFA Algorithms and Runtime Complexity

Recall from Section 5.1 that we use a pmatch structure to store computation state for pattern matches. A pmatch is associated with an event subsequence $e_0 \ldots e_k$, where e_0 and e_k are referred to as the StartEvent and EndEvent of the pmatch (their lifetimes are denoted by [StartLE, StartRE) and [EndLE, EndRE), respectively). We assume no ϵ-arcs in the AFA for simplicity.

Insert Algorithm Algorithm 1 shows the insertion algorithm that was described in Section 5.1; we cover additional details below.

If the inserted event e is out-of-order (with a timestamp t), we first call RemoveInvalidatedSequences (Lines 10—19) to delete the pmatch nodes that are no longer valid. We start at the rb-tree entry with next largest timestamp after t, and begin deleting pmatch nodes from the linked list until we reach a pmatch with

StartLE $> t$. If a deleted pmatch corresponds to a final state, we output a retraction event to compensate for the invalid prior output. We repeat the process until we reach an rbtree entry that contains no affected pmatch nodes. By invariant (Completeness), we can stop because if there were any future affected pmatch node, there would have been an affected pmatch in this entry. Thus, we avoid traversing pmatch entries that do not need to be deleted.

We next (Lines 3—7) retrieve the pmatch list for the previous event, and apply the next-ID relation $\vdash_M$ (with input e) to the ID corresponding to each pmatch. Each application of $\vdash_M$ to a pmatch node p returns a set of new pmatch nodes that are said to be *derivable from* p. This process returns a list L of pmatch nodes for sequences ending at e. We also attempt to start a new match (from q_0) beginning at event e and append this to L. If any reached state is final, we output an event. Event e is added to rbtree and associated with list L, which follows the (Ordering) invariant by construction.

The final step in case of out-of-order events (Lines 20—25), is to apply subsequent events (received previously) in rbtree to the matches in L. We continue this process until no new pmatch nodes get created. During this process, we can maintain the (Ordering) invariant without having to sort the pmatch lists (see Line 24).

Runtime Complexity Let each window hold at most w events, and let there be r entries in rbtree (r depends upon the number of events after the most recent punctuation). Assume that at most m partial matches can end at the same event[3], and each AFA transition takes constant time. Lines 3—7 perform rbtree lookup and transitions, incurring a total runtime of $O(\lg r + m)$. For an out-of-order event that is disordered by at most d events, RemoveInvalidatedSequences has runtime linear in the number of pmatch entries to be deleted, after the lookup in rbtree — this is $O(\lg r + \min(w, d) \cdot m)$. Finally, PropagateInsert has runtime linear in the number of new pmatch nodes created, which is $O(m)$. Thus, the algorithm has overall runtime $O(\lg r + \min(w, d) \cdot m)$ for an out-of-order event, or $O(\lg r + m)$ for an in-order event. Note that we pay a higher price only for out-of-order events, with low overhead if an event is only slightly out-of-order. Further, if we know a priori that the stream is ordered (e.g., due to a Cleanse operator upstream or from the data source), we can avoid the rbtree lookup, giving a runtime of $O(m)$.

Delete Algorithm It is possible that an upstream CQ deletes (retracts) an event that it issued earlier. Deletion of an event e proceeds by first invoking RemoveInvalidatedSequences (see Algorithm 1) with the timestamp of e, in order to delete all matches that depend on e, and issue the necessary output retractions. After removing this entry from rbtree, we look for new matches continuing from the entry before e, by invoking PropagateInsert for that entry.

C.1 State Cleanup using Punctuations

A punctuation along the event (or arc) input with timestamp t implies no future events (or arc-events) with timestamp less than t.

Effective Punctuation Let t_e and t_a denote the latest punctuations along the event and arc inputs respectively. For the purpose of AFA state cleanup, the *effective* incoming punctuation for the AFA operator is $t_p = \min(t_e, t_a)$. Note that $t_p = t_e$ in the static pattern case since $t_a = \infty$. To see why $t_p = \min(t_e, t_a)$, note that in case $t_a < t_e$, a subsequent arc-event with LE $\geq t_a$ can require the computation of AFA transitions using existing events with a timestamp of t_a or more. Under the default operation mode, the arc punctuation t_a is always implicitly t_{curr}, which makes the effective punctuation

[3]As in any NFA, $m = O(a^w)$ in the worst case, for at most a outgoing arcs per NFA state. Note that most automata in practice, such as those in Figures 3 and 14, have a constant m for partial matches starting from a given event; which results in $m = O(w)$, since we have to start a new matching process for each event in the stream.

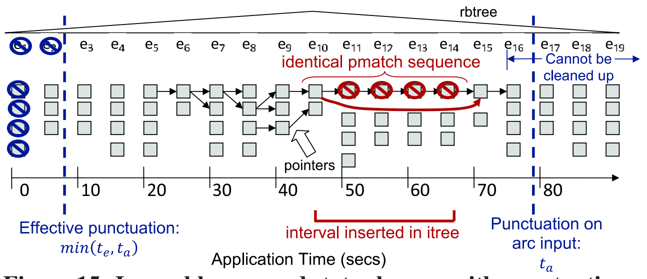

rbtree
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15 e16 e17 e18 e19
identical pmatch sequence
Cannot be cleaned up
pointers
0 10 20 30 40 50 60 70 80
Effective punctuation: $min(t_e, t_a)$
interval inserted in itree
Punctuation on arc input: t_a
Application Time (secs)

Figure 15: Ignorable arcs and state cleanup with punctuations.

$t_p = t_e$ instead of $min(t_e, t_a)$, since $t_e \leq t_a$. Thus, the effective punctuation is identical to the static pattern case.

Cleanup Algorithm Let cover(t) denote the latest event in rbtree with a timestamp earlier than t. We have the following invariant:

- (Cleanup) Let t_p be the effective punctuation timestamp. There exist no pmatch entries with EndEvent earlier than cover(t_p). Further, there exist no events with timestamp $< t$. Finally, there exist no arc-events with RE $\geq t_p$.

Given an effective punctuation with timestamp t_p, we traverse the rbtree from left to right, deleting all the pmatch entries and events, until we reach cover(t_p). This event is deleted, but its pmatch entries (and the entry in rbtree) are left untouched. Subsequent events and pmatch nodes are retained because out-of-order events may need to access them for applying transitions. Further, we can declare all output events with a timestamp before t_p as final, by sending out a punctuation t_p. Finally, an arc-event e_a can be discarded when the effective punctuation is e_a.RE or more.

Discussion Notice that we clean events and pmatch entries even if their constituent event lifetimes extend beyond the effective punctuation. Such aggressive cleanup is possible because the latest set of pmatch entries just before t *cover* all previous entries. In order words, since we know that there can be no new event with a timestamp before t, the earliest possible out-of-order event insertion will require looking up no earlier than cover(t). As noted in Section 5.4, our dynamic pattern semantics allow us to aggressively cleanup an event e as soon as the punctuation crosses e.LE.

D Supporting Ignorable Arcs

Pattern CQs should efficiently skip events that do not satisfy certain conditions [2]. For a particular pattern, only some events might be relevant, with the rest ignored to enable detection to continue. For instance, in query Q_1 (see Example 1), we may wish to ignore small price drops during an uptick phase (and vice versa).

An ignorable arc a_i is one that always translates into the next-ID relation $(e\alpha, q, z) \vdash_M (\alpha, q, z)$ when the fence function $f_i(e, z)$ is true. Thus, a_i is a self-loop with transfer function $g_i(e, z) = z$, and can be identified at query registration time. An AFA can operate correctly without special handling for ignorable events; however, this solution can lead to significant memory overhead, particularly in the presence of stream disorder. We need special optimization techniques to detect such transitions and treat them specially to avoid overhead. Interestingly, we will see that optimized handling of ignorable events can also significantly reduce the number of output events in the presence of disorder, and improve throughput.

Solution Overview Recall that a pmatch node for a subsequence $e_0 \dots e_k$ contains 4 fields: StartLE $= e_0$.LE, StartRE $= e_0$.RE, q, and r. We observe that a sequence of consecutive transitions along an ignorable arc results in the creation of identical pmatch nodes in rbtree, which are stored in consecutive rbtree entries. We leverage this observation by creating an interval tree called *itree* in association with rbtree. Every maximal consecutive sequence of identical

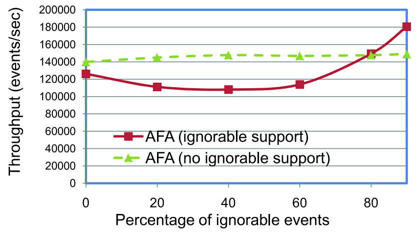


Figure 16: Throughput vs. ignorable.

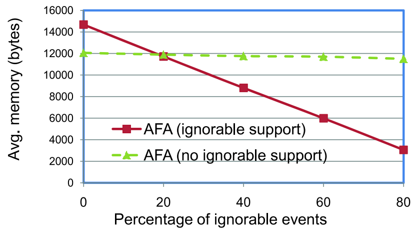


Figure 17: Memory vs. ignorable.

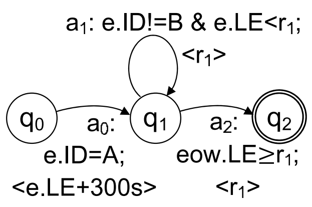


Figure 18: Negative pattern.

pmatch nodes, $p_1, \ldots, p_j$, where $p_1 \vdash_M p_2 \vdash_M \ldots \vdash_M p_j$, is replaced by (1) a single pmatch node p_1 in rbtree (deleting the subsequent identical nodes), and (2) an interval $(p_1.\text{EndLE}, p_j.\text{EndLE}]$ in itree that indicates the time interval over which p_1 repeats itself, and points to p_1. Node p_1 is called an *anchor* node.

Algorithmic Changes to Handle Ignorable Arcs To each pmatch node p_i, we add (1) a pointer to its previous pmatch node, (2) a pointer to each next pmatch node derivable from p_i either directly or via a consecutive sequence of ignorable deleted pmatch nodes, and (3) a pointer to an associated itree interval, if p_i is an anchor.

Example 6 (Ignorable Arcs)**.** *Figure 15 shows an rbtree with pmatch nodes and extra pointers to represent next-ID derivations. The identical pmatch nodes (between events e_{11} and e_{14}) are removed, leaving only the anchor at e_{10} with a pointer to its indirect next derivation at e_{15}. The corresponding itree entry (interval) is also shown.*

When an event e arrives with timestamp t:

1. We retrieve the set S_1 of immediately preceding pmatch nodes via $\text{Search}_{\leq}(t)$. We also lookup itree for intervals stabbed by the EndLE of these pmatch nodes; this lookup returns a set S_2 of affected anchor pmatch nodes.
2. We apply ignorable arcs to each node p in $S = S_1 \cup S_2$, and denote the set of successful nodes by S_3. The current derivations of $p \in S_3$ can be left untouched since e does not invalidate any of them. If p is already an anchor, its interval is extended if necessary; otherwise, we convert it into an anchor and add the corresponding interval in itree.
3. For each node p in $(S - S_3)$, we iteratively invalidate all their derived pmatch nodes (by following the next pointers). If p is an anchor, the right endpoint of the interval is truncated to t.
4. Finally, MakeTransitions (for non-ignorable arcs only) is invoked for the nodes in S, followed by PropagateInsert for the new pmatch nodes.

The worst-case per-event overhead of our scheme is $O(k \lg k)$, where k is the number of maximal sequences of identical pmatch nodes. Delete proceeds similarly and is omitted for brevity. Cleanup using a punctuation t_p proceeds as usual; however, an interval in itree and the associated pmatch anchor can be cleaned up only when $\text{cover}(t_p)$ lies after the right endpoint of the interval.

Avoiding Frequent itree Updates In the common case of in-order events that cause the buildup of an increasingly long sequence of identical pmatch nodes, we want to avoid updating itree after every event. To solve this problem, when we receive an in-order event that first triggers an ignorable arc, we associate the anchor p_1 with the interval $(p_1.\text{EndLE}, \infty)$ in itree. Thus, if subsequent events are part of the same sequence, itree does not have to be updated. When a new event e does not trigger the ignorable arc, the sequence ends and we truncate the interval to $(p_1.\text{EndLE}, e.\text{EndLE})$.

Discussion When we receive an out-of-order event that performs an ignorable transition, we do not invalidate and re-build the spanning matches. This helps reduce chattiness at the output (see Figure 10), and even improves throughput when the percentage of ignorable events is high (see below). Finally, we note that the events contributing to ignorable arcs are not deleted; they may be required to compute transitions due to out-of-order events and arc-events.

Evaluation We use the AFA+ignore operator with CQ Q_1 of Section 6 ($k = 1, p = 0.5, W = 500, \delta = 0$) and increase the percentage of ignorable events I (we use an ordered stream). Figures 16 shows that throughput is generally increasing because increasing I implies fewer matches, which the technique optimizes for. On the other hand, we see a slight drop in throughput at the start, due to the overhead of looking up itree. Beyond $I = 80\%$ our technique gives higher throughput than the basic operator without the optimization. Figure 17 shows that memory usage drops significantly as I increases, when we use the ignorable optimization. Without the optimization, memory usage remains high as we increase I even though it is slightly better when $I < 20\%$ due to the overhead of maintaining pointers to itree entries.

E Handling Negative Patterns with AFAs

A common requirement for pattern CQs is to handle the *absence* of certain subpatterns, as part of the overall pattern. One way to handle negative patterns is to first generate positive matches and then prune the negative matches. However, we wish to handle negative patterns without generating intermediate false-positive results.

AFAs are can directly accept patterns with positive and negative subpatterns, and provide an efficient execution layer for pattern languages that can express negation [2, 28]. The case where a negative subpattern appears between positive subpatterns is easily handled by a single AFA. Consider the more complicated corner case where the pattern ends with a negative subpattern. For example, we want to detect a trade for stock A, followed by no trades of a competing stock B within $w = 300$secs. This query can be written using the AFA in Figure 18. The register $\langle r_1 \rangle$ tracks the timeout for detecting the absence of stock B, and a_2 is a special arc that is triggered by an end-of-window (*eow*) indicator (this can be implemented, for example, using punctuations). Note that this corner case can alternately be handled using a separate *anti-semijoin* stream operator.

F Aggressive State Cleanup in AFAs

Figure 15 shows an rbtree. The left line is the effective punctuation $t_p = \min(t_e, t_a)$, i.e., the minimum of the punctuations along the event and arc inputs. The right line is t_a. Recall that $\text{cover}(t)$ denotes the latest event with a timestamp earlier than t. Items at $\text{cover}(t_a)$.LE and later cannot be cleaned up because an arbitrary arc may be inserted in future. Items earlier than $\text{cover}(t_p)$.LE can be cleaned up as depicted in Figure 15 (see Section 5.1). We now investigate cleanup between these two timestamps.

F.1 Aggressive Event Deletion

Consider the special case where (1) the fence function $f_i(\bar{E}, \bar{R})$ associated with an arc a_i is expressible as a conjunction $f_i^{\bar{E}}(\bar{E}) \wedge f_i^{\bar{R}}(\bar{R})$, and (2) the transfer function $g_i(\bar{E}, \bar{R})$ is reducible to $g_i^{\bar{R}}(\bar{R})$, i.e., it is computable only over the previous register.

Algorithm 2: Computing path punctuations.

```
1 ComputePathPunctuation(path $p = a_1 \ldots a_k$) begin
2     $\bar{\pi} \leftarrow \pi_1$;
3     foreach arc $a_j$ in ordered path $p$ do
4         $e \leftarrow$ earliest event s.t. $e.\mathsf{LE} \geq \bar{\pi}$ and $f_j^{\bar{E}}(e)$ is true;
5         $t \leftarrow \min(e.\mathsf{LE}, \pi_j)$;
6         $\bar{\pi} \leftarrow \max(\bar{\pi}, t)$;
7     return $\min(\bar{\pi}, t_a)$;
8 end
```

We define the *triggering set* of an event e as the set of arcs a_i such that $f_i^{\bar{E}}(e)$ is true. If events are large in size, we can delete an event e and replace the event pointer in rbtree with its triggering set L (note that this requires computing $f_i^{\bar{E}}(e) \forall a_i \in \mathcal{A}$). This optimization is possible because we no longer need e to determine if an arc a_i is triggered — we only need to check whether $a_i \in L$, and if yes, we apply the fence function $f_i^{\bar{R}}(r)$, where r is the previous register value. Further, if a_i is triggered, we compute the new register value by invoking the transfer function $g_i^{\bar{R}}(r)$.

F.2 Leveraging Punctuations with Predicates

Assume that (1) the fence function $f_i(\bar{E}, \bar{R})$ associated with an arc a_i is expressible as a conjunction $f_i^{\bar{E}}(\bar{E}) \wedge f_i'(\bar{E}, \bar{R})$, and (2) there are no retractions in the input stream to the AFA operator.

We can leverage predicated punctuations to clean state. A *predicated punctuation*, also called a *partial order guarantee* [17], is associated with a timestamp t and a condition C, and is a guarantee that no event arriving in the future and satisfying C can have a timestamp earlier than t. Predicated punctuations may be inserted when performing a union across multiple streams, by a data source, based on application semantics [26], or by network protocols [17].

The basic idea is to use predicated punctuations and the AFA graph structure to determine what additional registers and events can be deleted. An *arc punctuation* for an arc a_i is the largest timestamp π_i with a guarantee that no event e arriving in the future and for which $f_i^{\bar{E}}(e)$ is true, can have a timestamp earlier than π_i. We can easily use the set of predicated punctuations to infer an arc punctuation for every arc in the AFA. For example, assume that a stream contains a union of sensor readings across multiple floors. If an arc a_i has the fence condition $f_i^{\bar{E}}(\bar{E}) = \{\mathsf{Floor} = 3 \wedge \mathsf{Temperature} > 95\}$ and we have a predicated punctuation with timestamp 20 and condition $\{\mathsf{Floor} \leq 3\}$, we can infer that $\pi_i = 20$.

An *path punctuation* with timestamp $\bar{\pi}(p)$ for a path $p = a_1 \ldots a_k$ in the AFA directed graph is a promise that no future out-of-order event with timestamp earlier than $\bar{\pi}(p)$ can trigger a_1 and then cause the execution of all subsequent transitions along path p.

Computing Path Punctuations As a first step, $\bar{\pi}(p)$for a path $p = a_1 \ldots a_k$ can be set to π_1, since clearly no future event with timestamp earlier than $\bar{\pi}_1$ can trigger a_1. We can do better using Algorithm 2 which leverages the AFA structure. Consider the path $p = a_1 \ldots a_k$. We start with $\bar{\pi}(p) = \pi_1$. In Lines 3–6, for each arc a_j on the path p, we look at the existing events and π_j to try and push $\bar{\pi}(p)$ ahead (Lines 4 and 5) to the latest possible timestamp $t \geq \bar{\pi}(p)$ such that a traversal of arc a_j at timestamp earlier than t is not possible. Intuitively, the non-existence of a later event e that can cause transition a_j implies that a match traversing the path from a_1 to a_j is not possible. Note that we can use memoization to share partial results while computing $\bar{\pi}(p)$ for many paths in the graph.

Cleaning State Recall that each AFA register is associated with a pmatch node in some AFA state. Consider each non-final state q in turn. Let t_1 denote the minimum $\bar{\pi}(p)$ across the set of unique paths p in M from q to some final state, where uniqueness is determined by the set of edges in p. We can delete all registers corresponding to pmatch entries (in rbtree) that are associated with q and that lie to the left of $\mathsf{cover}(t_1)$ in rbtree (i.e., whose EndLE is less than $\mathsf{cover}(t).\mathsf{LE}$). Furthermore, for every event e, let t_2 be the minimum $\bar{\pi}(p)$ across all unique paths to a final state that contain (but do not begin with) some arc in e's triggering set. We can delete e if its timestamp is earlier than t_2.

G Additional Details on Related Work

Stream Pattern Matching The first generation of pattern CQs support regular expressions over streams [16, 17, 19, 28, 35]. These patterns are less expressive than AFA, and existing proposals mostly use deterministic finite automata based algorithms that can explode an NFA to an exponential number of states in the presence of non-determinism, which is common in stream pattern CQs.

ZStream [20] supports sequence queries with aggregates, but registers carry state only within an *operator* (such as Kleene closure). At the expense of lower expressiveness (e.g., this model cannot express $a^i \cdot b^i$), ZStream supports new operator re-ordering optimizations. As discussed earlier, other research [2, 10] allows the detection of more complex static patterns over ordered streams using the CAN model. We continue further in this direction, and support dynamic patterns over disordered streams with revisions, under an execution model with unconstrained arcs. Multi-query optimization schemes proposed earlier [10, 11] are directly applicable to us, particularly in the context of dynamic patterns. The *match buffer* [28] used to report the events that contribute to a pattern can be adapted by our AFA to output the set of events contributing to a detected match. We also note that pattern languages such as SASE+ [28] can be compiled into an AFA.

Disorder and Dynamic Patterns The initial generation of streaming systems assumed that events arrive in-order at the DSMS [4]. One solution to handle disorder is k-slack [5], where the stream is assumed to be disordered by at most k tuples or time units, with reordering performed before stream processing. Such an approach can lead to higher latency, particularly in the presence of ignorable events. The ability to issue compensations using retractions is provided by several systems; examples include revision tuples [23], Dstreams [21], and lifetime modifications [6]. Out-of-order processing has also been proposed for regular expressions [16], sequence queries [17], and relational operators [6, 32], but these techniques do not directly apply to rich patterns. We apply partial order guarantees [17] to clean state by introducing path punctuations.

Our earlier work describes techniques to detect forward progress using recursive queries with relational operators [7]. Recursive queries can express patterns succinctly, but are less efficient and require modifications to the DSMS engine. In contrast, we provide native out-of-order support for a powerful AFA-based execution model of wide applicability, with extensions to handle several requirements. Finally, dynamic patterns are reminiscent of systems such as PSoup [29] that treat queries and data symmetrically.

Additional References

[29] S. Chandrasekaran and M. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2), 2003.

[30] Head & Shoulders Pattern. `http://tinyurl.com/6e6qtb`.

[31] S. Jeffery et al. Declarative support for sensor data cleaning. In *Pervasive*, 2006.

[32] J. Li et al. Out-of-order processing: a new architecture for high-performance stream systems. In *VLDB*, 2008.

[33] Snort Network Intrusion Detection System. `http://snort.org`.

[34] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 1970.

[35] F. Yu et al. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, 2006.