Towards Expressive Publish/Subcribe Systems

Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White

Cornell University, Department of Computer Science
{ademers,johannes,mshong,mirek,wmwhite}@cs.cornell.edu

Abstract. Traditional content based publish/subscribe (pub/sub) systems allow users to express stateless subscriptions evaluated on individual events. However, many applications such as monitoring RSS streams, stock tickers, or management of RFID data streams require the ability to handle *stateful* subscriptions. In this paper, we introduce Cayuga, a stateful pub/sub system based on nondeterministic finite state automata (NFA). Cayuga allows users to express subscriptions that span multiple events, and it supports powerful language features such as parameterization and aggregation, which significantly extend the expressive power of standard pub/sub systems. Based on a set of formally defined language operators, the subscription language of Cayuga provides non-ambiguous subscription semantics as well as unique opportunities for optimizations. We experimentally demonstrate that common optimization techniques used in NFA-based systems such as state merging have only limited effectiveness, and we propose novel efficient indexing methods to speed up subscription processing. In a thorough experimental evaluation we show the efficacy of our approach.

1 Introduction

Publish/Subscribe is a popular paradigm for users to express their interests ("subscriptions") in certain kinds of events ("publications"). Traditional publish/subscribe (pub/sub) systems such as topic-based and content-based pub/sub systems allow users to express stateless subscriptions that are evaluated over each event that arrives at the system; and there has been much work on efficient implementations [14]. However, many applications require the ability to handle *stateful* subscriptions that involve more than a single event, and users want to be notified with customized witness events as soon one of their stateful subscriptions is satisfied. Let us give two example applications that motivate the types of stateful subscriptions that a *stateful* pub/sub system needs to handle.

Example 1: Stock Ticker Event Monitoring. Consider a system that permits financial analysts to compose subscriptions over a stream of stock ticks [2]. Some sample subscriptions are shown in Table 1. Subscription S1 is a traditional pub/sub subscription, and it can be evaluated on each incoming event individually. However, an important feature of event processing systems is the ability to detect specific *sequences* of events, as shown in the next four subscriptions. To detect sequences, the system has to maintain *state* about events that have previously entered the system. For example, to process Subscription S2, the system has to "remember" whether an event with a stock

Subscription	Description
S1	Notify me when the price of IBM is above \$100.
S2	Notify me when the price of IBM is above \$100, and the *first* MSFT price afterwards is below \$25.
S3	Notify me when there is a sale of some stock at some price (say p), and the next transaction is a sale of the same stock at a price above $1.05 \cdot p$.
S4	Notify me when the price of any stock increases monotonically for $\geq$ 30 minutes.
S5	Notify me when the next IBM stock is above its 52-week average.
S6	Once military.blog.com posts an article on US troop morale, send me the first post referencing (i.e., containing a link to) this article from the blogs to which I subscribe.
S7	Send postings from all blogs to which I subscribe, in which the first posting is a reference to a sensitive site XYZ, and each later posting is a reference to the previous.

Table 1. Sample Subscriptions

price of IBM above \$100 has happened since the most recent MSFT event; only then are we interested in learning about future MSFT prices. Subscriptions S3 and S4 illustrate another important component: We need to support parameterized subscriptions, i.e., subscriptions that contain parameters that are bound at run-time to values seen in events. As an example, in Subscription S3, we are looking for *some* stock that exhibits a 5% jump in price; instead of having to register a subscription for each possible stock symbol, we register a single subscription with a *parameter* that is set at run time. Subscription S4 requires support for *aggregation*, and Subscription S5 is an example that combines both parameterization and aggregation.

Example 2: RSS Feed Monitoring. Our second motivating application is online RSS Feed Message Brokering. RSS feeds have become increasingly important for online exchange of news and opinions. With a stateful pub/sub system, users can monitor RSS Feeds and register complex subscriptions that notify the users as soon as their requested RSS message sequences have occured. Subscriptions S6 and S7 in Figure 1 are examples in this domain.

To reiterate: Traditional pub/sub systems scale to millions of registered subscriptions and very high event rates, but have limited expressive power. In these systems, users can only submit subscriptions that are predicates to be evaluated on single events. Any operation across multiple events must be handled externally. In our proposed stateful pub/sub system, however, subscriptions can span multiple events, involving parameterization and aggregation, while maintaining scalability in the number of subscriptions and event rate. In comparison, full-fledged Data Stream Management Systems (DSMS) [3, 23, 12] have powerful query languages that allow them to express much more powerful subscriptions than stateful pub/sub systems; however, this limits their scalability with the number of subscriptions, and existing DSMSs only do limited multi-query optimization. Figure 2 illustrates these tradeoffs.

Another area very closely related to stateful pub/sub is work on *event systems*. Event systems can be programmed in languages (called *event algebras*) that can compose complex events from either basic or complex events arriving online. However, we have observed an unfortunate dichotomy between theoretical and systems-oriented approaches in this area. Theoretical approaches, based on formal languages and well-defined semantics, generally lack efficient, scalable implementations. Systems approaches usually lack a precise formal specification, limiting the opportunities for query optimization and query rewrites. Indeed, previous work has shown that the lack of clean operator semantics can lead to unexpected and undesirable behavior of complex algebra expressions [15, 28]. Our approach was informed by this dichotomy, and we have taken great care to define a language that can express very powerful subscriptions, has a precise formal semantics, and can be implemented efficiently.

Our Contributions. In this paper, we propose Cayuga, a stateful publish/subscribe system based on a nondeterministic finite state automata (NFA) model. We start by introducing the Cayuga event algebra, which can express all example subscriptions shown in Table 1, and we illustrate how algebra expressions map to linear finite sate automata with self-loops and buffers (Section 2). To the best of our knowledge, this is the first work that combines a formal event language definition with a methodology to efficiently implement the language. We then overview the implementation of our system which leverages techniques from traditional pub/sub systems as well as novel MQO techniques to achieve scalability (Section 3). In a thorough experimental study, we evaluate the scalability of our system both with the number of subscriptions and their complexity, we evaluate the efficacy of our MQO techniques, and we show the performance of our system with real data from our two example application domains (Section 4). We discuss related work in Section 5, and conclude in Section 6.

In closing this introduction, we would like to emphasize two important aspects of our approach. First, instead of adding features to a pub/sub system in an ad-hoc fashion, our system is based on formal language operators and therefore provides unambiguous query semantics that are necessary for query optimization. Second, compared to similar approaches that use NFAs for scalability such as YFilter [13], Cayuga supports powerful language features such as parameterization and aggregation that require us to extend beyond prior work. One interesting result from our experimental study is that common optimization techniques used in NFA-based systems, such as state merging, have only limited effectiveness for the workloads that we consider. On the other hand, some of our novel MQO techniques can potentially be applied to other NFA-based systems.

2 Cayuga Algebra and Automaton

2.1 Data Model

Our event algebra consists of a data model for event streams plus operators for producing new events from existing events. An event stream, denoted as S or S_i, is a (possibly infinite) set of event tuples $\langle \overline{a}; t_0, t_1 \rangle$. As in the relational model, $\overline{a} = (a_1, \ldots, a_n)$ are data values with corresponding attributes (symbolic names). The t_i's are temporal values representing the start (t_0) and end timestamps (t_1) of the event. For example, in the stock monitoring application, assume the stream of stock sales published by the data source has fields $(\texttt{name}, \texttt{price}, \texttt{vol}; \texttt{timestamp})$. An event from that stream then could be the tuple $\langle \text{IBM}, 90, 15000; \text{9:10}, \text{9:10} \rangle$. The timestamps are identical, because each sale is an instantaneous event. We assume each event stream has a fixed schema, and events arrive in temporal order. That is, event e_1 is processed before e_2

Algebraic Expressions
S1: $\sigma_\theta(S_1)$, where $\theta = S_1.\texttt{name} = \text{IBM} \wedge S_1.\texttt{price} > 100$
S2: $\sigma_{\theta_2}(\sigma_\theta(S_1) ;_{\theta_1} S_2)$, where θ same as in Subscription S1, $\theta_1 = S_2.\texttt{name} = \text{MSFT}, \theta_2 = S_2.\texttt{price} < 25$
S3: $\sigma_{\theta_2}(S_1 ;_{\theta_1} S_2)$, where $\theta_1 = S_2.\texttt{name} = S_1.\texttt{name}, \theta_2 = S_2.\texttt{price} > 1.05 * S_1.\texttt{price}$
S4: $\sigma_{\theta_3}(\mu_{\sigma_{\theta_2},\theta_1}(S_1, S_2))$, where $\theta_1 = S_2.\texttt{name} = S_1.\texttt{name}, \theta_2 = S_2.\texttt{price} >= S_2.\texttt{price.last}, \theta_3 = \text{DUR} >= 30min$
S5: $\sigma_{\theta_2}(\mathcal{E} ;_{\theta_1} S_3)$, where $\mathcal{E} = \sigma_{\text{DUR}=52\text{ weeks}}\left(\mu_{\alpha_{g_2},\text{TRUE}}(\alpha_{g_1} \circ \sigma_\theta(S_1), \sigma_\theta(S_2))\right)$, $\theta = \texttt{name} = \text{IBM}, \theta_1 = S_3.\texttt{name} = \text{IBM}, \theta_2 = S_3.\texttt{price} > \text{AVG}$
S6: $\sigma_{\theta_1}(S_1) ;_{\theta_2} \sigma_{\theta_3}(S_2))$, where $\theta_1 = S_1.\texttt{website} =$ 'military.blog.com'$\wedge S_1.\texttt{category} =$ 'US troop morale', $\theta_2 = contains(S_2.\texttt{description}, S_1.\texttt{link}), \theta_3 = (S_2.\texttt{website} = site_1 \vee \ldots S_2.\texttt{website} = site_n)$
S7: $\mu_{\text{ID},\theta_1}(\sigma_{\theta_3 \wedge \theta_2}(S_1), \sigma_{\theta_3}(S_2))$, where $\theta_1 = contains(S_2.\texttt{description}, S_2.\texttt{link.last})$, $\theta_2 = contains(S_1.\texttt{description}, `XYZ')$, θ_3 same as in Subscription S6

Table 2. Algebraic Expressions

iff $e_1.t_1 \leq e_2.t_1$. However, a stream may contain events with non-zero duration, overlapping events and simultaneous events (events with identical time stamp values). Our operator definitions depend on the timestamp values, so we do not allow users to query or modify them directly. However, we do allow constraints on the *duration* of an event, defined as $t_1 - t_0 + 1$ (we treat time as discrete, so the duration of an event is the number of clock ticks it spans). We store starting as well as ending timestamps and use interval-based semantics to avoid well-known problems involving concatenation of complex events [15].

2.2 Operators

Our algebra has four unary and three binary operators. Due to space constraints, we give here only a brief description of them here; a formal definition and more examples can be found in our technical report [1].

The first three unary operators, the **projection** operator $\pi_{\mathbb{X}}$, the **selection** operator σ_θ, and the **renaming** operator ρ_f are well known from relational algebra. Projection and renaming can only affect data values; temporal values are always preserved. As the renaming operator only affects the schema of a stream and not its contents, we will often ignore this operator for ease of exposition. Instead, we will denote attributes of an event using the input stream and a dot notation, making renaming implicit. For example, the `name` attribute of events from stream S_1 will be referred to as $S_1.\texttt{name}$. A selection formula is any boolean combination of atomic predicates of the form τ_1 relop τ_2, where the τ_i are arithmetic combinations of attributes and constants, and relop can be one of $=, \leq, <, \geq, >$, or string matching. We also allow predicates of the form DUR relop c where the special attribute DUR denotes event duration and c is a constant. The unary operators above enable filtering of single events and attributes, equivalent to a classical pub/sub system. Subscription S1 is an example of such a stateless subscription.

The added expressive power of our algebra lies in the binary operators, which support subscriptions over multiple events. All of these operators are motivated by a corresponding operator in regular expressions. The first binary operator is the standard

union operator $\cup$, where $S_1 \cup S_2$ is defined as $\{\, e \mid e \in S_1 \text{ or } e \in S_2 \,\}$. Our second operator is the **conditional sequence** operator $S_1 \,;_\theta\, S_2$. For streams S_1 and S_2, and selection formula θ (a predicate), $S_1 \,;_\theta\, S_2$ computes sequences of two consecutive and non-overlapping events, filtering out those events from S_2 that do not satisfy θ. Adding this feature is essential for parameterization, because θ can refer to attributes of both S_1 and S_2. This enables us to express "group-by" operations, e.g., to group stock quotes by name via $S_1 \,;_\theta\, S_2$, with θ being $S_1.\texttt{name} = S_2.\texttt{name}$. $S_1 \,;_\theta\, S_2$ essentially works as a join, combining each event in S_1 with the event immediately after it in S_2. However, θ works as a filter, removing uninteresting intervening events. Subscriptions S2 and S3 are examples of such subscriptions.

Our third binary operator is the **iteration** operator $\mu_{\mathfrak{F},\theta}(S_1, S_2)$, motivated by the Kleene-+ operator. Informally, we can think of $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ as a repeated application of conditional sequencing: $S_1 \cup (S_1 \,;_\theta\, S_2) \cup (S_1 \,;_\theta\, S_2 \,;_\theta\, S_2) \cup \cdots$. Each clause separated by the $\cup$ operator corresponds to an iteration of processing an event from S_2 which satisfies θ. The additional parameter $\mathfrak{F}$, a composition of selection, projection and renaming operators, enables us to modify the result of each iteration. Thus μ acts as a fixed point operator, applying the operator $;_\theta$ on each incoming event repeatedly until it produces an empty result. To avoid unbounded storage, at each interation, it will only remember the attribute values from stream S_1 and the values from the most recent iteration of S_2. For any attribute ATT_i in S_2, we refer to the value from the most recent iteration via $\text{ATT}_i.\texttt{last}$. Initially, this value is equivalent to the corresponding attribute in S_1, but it will be overwritten by each iteration.

At first it might seem surprising that our algebra needs $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ to express the equivalent of something as simple as $(S_2)^+$ in regular languages. The reason, like for the $;_\theta$ operator, is that we want to support parameterization efficiently. In fact, θ serves the same purpose as in $;_\theta$: during each iteration it filters irrelevant events from S_2 when the *next* event from S_2 is selected. In Subscription S5, it was used to make sure that no quotes for other companies would be selected for a sequence of IBM prices, and vice versa. Similarly, $\mathfrak{F}$ removes irrelevant events during each iteration, like non-increasing sequences in the example. Another interesting feature is that μ is a binary operator, while Kleene-+ is unary. One reason, as can be seen in the definition of μ, is that we need a way to initialize our attributes $\text{ATT}_i.\texttt{last}$. The other reason is that, by adding S_1 to μ, both $\mathfrak{F}$ and θ can refer to S_1's attributes. This enables us to support powerful parameterized subscriptions such as S4.

Aggregates fit naturally into our algebra, where aggregation occurs over a sequence of events. Our **aggregate** operator is α_g, where g is a function used to introduce a new attribute to the output. Together with μ, we get a natural aggregate of the form $\alpha_{g_3}\big(\mu_{\alpha_{g_2} \circ \mathcal{F},\theta}(\alpha_{g_1}(\mathcal{E}_1), \mathcal{E}_2)\big)$. In this expression, α_{g_1} functions as an initializer, α_{g_2} is an accumulator, and α_{g_3} is a finalizer. For example, suppose we want the average of IBM stock over the past 52 weeks, as referenced in Subscription S5. If we let S_1, S_2, S_3 all refer to our stream of stock quotes, S, this is expressed as $\mathcal{E} = \sigma_{\text{DUR}=52\text{ weeks}}\big(\mu_{\alpha_{g_2},\text{TRUE}}(\alpha_{g_1} \circ \sigma_\theta(S_1), \sigma_\theta(S_2))\big)$, where θ is $\texttt{name} = \text{IBM}$, g_1 is defined as $\text{AVG} \mapsto \texttt{price}$, $\text{COUNT} \mapsto 1$, and g_2 is defined as $\text{AVG} \mapsto \frac{\text{COUNT}.\texttt{last} \times \text{AVG}.\texttt{last} + \texttt{price}}{\text{COUNT}.\texttt{last} + 1}$, $\text{COUNT} \mapsto \text{COUNT}.\texttt{last} + 1$. Notice that we use the $\texttt{last}$ feature of μ to compute our aggregate recursively. The average is now a value attached to an attribute and can be used by the remaining part of Subscription S5. Therefore Subscription S5 can be expressed as $\sigma_{\theta_2}(\mathcal{E}\,;_{\theta_1} S_3)$ where $\mathcal{E}$ is defined above, θ_1 is $S_3.\texttt{name} = \text{IBM}$, and θ_2 is $S_3.\texttt{price} > \text{AVG}$.

For completeness, Table 3 also contains the two RSS subscriptions listed in Table 1. Here we assume all the blogs the user subscribes to consist of $site_1, \cdots, site_n$, and $contains(T, P)$ is the substring match operator that tries to find substring pattern P in text T; ID is the identity function that has no effect on the input.

2.3 Automaton Description

Given the algebra's similarity to regular expressions, finite automata would appear to be a natural implementation choice. Similar to the classic NFA model, for an incoming event, an automaton instance in one state can explore all the out-going edges, and non-deterministically traverse any number of them. If it cannot traverse any edge, however, this instance will be *dropped*.

We extend standard finite automata [19] in two ways. First, attributes of events can have infinite domains, e.g., text attributes, and therefore the input alphabet of our automaton, which is the set of all possible events, can be infinite as well. To handle this case, we associate each automaton edge with a *predicate*, and for an incoming event, this edge is traversed iff the predicate is satisfied by this event. Second, to be able to generate customized notification and to handle parameterized predicates over infinite domains, we need to store in each automaton instance the attributes and values of those events that have contributed to the state transition of this instance. These attributes and values are called *bindings*. To avoid overwriting the bindings of earlier events with that of latter events, we also need an attribute renaming function for each edge so that when an event makes an automaton instance traverse that edge, the bindings in that event are properly renamed before being stored in the instance.

We have developed a mechanical way to translate algebra expressions into automata. Details of this mechanism as well as the proof of correctness can be found in our technical report [1]. Intuitively, for a given algebra expression, we first construct a parse tree, and then translate each tree node corresponding to a binary operator into an automaton node. In our mechanism any left-deep parse tree can be translated into a single automaton, referred to as a *left-deep* automaton. In the following sections, we focus only on left-deep expressions and automata. General algebra expressions can be handled by having several left-deep automata feed their (complex) output events to other left-deep automata [1].

We first use an example to illustrate a left-leep automaton. Let subscription AutQ be "Notify me when for any stock s, there is a monotonic decrease in price for at least 10 minutes, which starts at a large trade ($\texttt{vol} > 10,000$). The immediately next quote on the same stock after this monotonic sequence should have a price 5% above the previously seen (bottom) price." Its algebra expression is $\sigma_{\theta_5}(\sigma_{\theta_4}(\mu_{\sigma_{\theta_3},\theta_2}(S_1, S_2))\,;_{\theta_2} S_3)$. The S_i are shorthand notation for appropriately renamed and projected versions of S: $S_1 \equiv \rho_{f_1} \circ \pi_{\texttt{name,price}} \circ \sigma_{\theta_1}(S), S_2 \equiv \rho_{f_2} \circ \pi_{\texttt{name,price}}(S), S_3 \equiv \rho_{f_3} \circ \pi_{\texttt{name,price}}(S)$. The corresponding predicates and renaming functions are: $\theta_1 \equiv \texttt{vol} > 10,000, \theta_2 \equiv \texttt{company} = \texttt{company.last}, \theta_3 \equiv \theta_2 \wedge \texttt{minP} < \texttt{minP.last}, \theta_4 \equiv \theta_3 \wedge \text{DUR} \geq 10\,\text{min}, \theta_5 \equiv \theta_2 \wedge \texttt{price} >$

		Number of concurrent subscriptions	
		few	many
Complexity of subscriptions	low	(boring)	pub/sub
	high	DSMS	stateful pub/sub

Fig. 1. Tradeoffs between pub/sub and Data Stream Management Systems

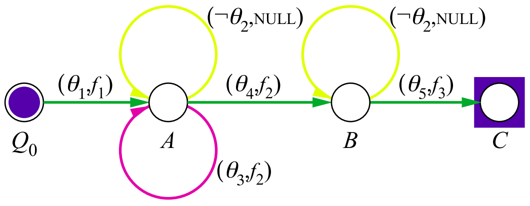


Fig. 2. Automaton for query AutQ

$1.05\,\texttt{minP}, f_1 \equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{maxP}), f_2 \equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{minP}), f_3 \equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{finalP})$. The explicit use of renaming is necessary for this example to make the schemas of the intermediate results at the different automaton nodes clear. The corresponding automaton is shown in Figure 1.

As opposed to NFA's with arbitrary structures, certain regularity is enforced by the translation from Cayuga algebra expressions. Now we describe some important properties of the structure of a left-deep automaton. Note that our MQO techniques described in Section 3 have a crucial dependence on these properties.

Each left-deep automaton is acyclic, except for self-loops. There are three types of edges, described as follows. *Forward* edges are those edges whose destination node is different from the source node, e.g., the edge from A to B in the example. Each node has at least one of them, except for the end node. Also on each node other than the start node, there will be two self-loop edges called *filter* and *rebind* edge respectively. We draw a filter edge on top of the node, a rebind edge below the node (see node A in Fig. 1). The predicate on a filter edge (or *filter predicate*) corresponds to the negation of the filter formula θ in $;_\theta$ or $\mu_{\mathfrak{F},\theta}$. Nodes A and B in Figure 1 are two examples of nodes containing filter edges that are translated from operators $\mu_{\mathfrak{F},\theta}$ and $;_\theta$ respectively. Also, by construction θ will appear in the forward and rebind edges of the same node as a conjunction to the remaining predicate there. Predicate θ_4 on the forward edge between node A and B in Figure 1 illustrates this. The reason for this automaton construction from algebra operators is that on the algebra side, an event is filtered when θ is not satisfied (or $\neg\theta$ is satisfied), and on the automaton side, this happens if it traverses the filter edge (and therefore cannot traverse any forward/rebind edge). Filter edges are unique among the three types of edges in that the traversal of a filter edge does not modify the bindings of the instance. If a node is not translated from $;_\theta$ or $\mu_{\mathfrak{F},\theta}$, the filter predicate will be FALSE, and we omit drawing the edge. A rebind predicate corresponds to the selection formula in $\mathfrak{F}$ of $\mu_{\mathfrak{F},\theta}$. Similarly, if a node is not translated from $\mu_{\mathfrak{F},\theta}$, the rebind predicate is FALSE, and we omit drawing the edge. The construction of rebind edge is illustrated in Figure 1 by node A, translated from $\mu_{\sigma_{\theta_3},\theta_2}$. Node B is shown without rebind edge since it is translated from operator $;_{\theta_2}$.

3 Implementation and MQO Techniques

Our algebra and automaton model are designed to be amenable to multi-query optimization. An obvious optimization is to merge equivalent states that occur in several automata. This is the approach taken by YFilter [13] and hence we omit the details.

The result of the merging process is a DAG with a single start node. In the following we focus on implementation challenges that are unique to Cayuga. For this discussion we need some additional notation.

3.1 Notation

A *static predicate* is a conjunction of atomic predicates that compare attribute values of the incoming event to constants, e.g., $\texttt{name} = \text{IBM} \wedge \texttt{price} > 10$. A *dynamic predicate* (or *parameterized* predicate) is a conjunction of atomic predicates of the form $\textsc{att}_1$ relop $\textsc{att}_2$, which compares an attribute value of the incoming event with an attribute of an earlier event. An example is θ_2 in Subscription S3.

For ease of exposition, in the following discussion we assume that each predicate is a conjunction of atomic predicates. Our techniques can be easily generalized to arbitrary boolean combinations of atomic predicates by requiring that predicates be supplied in disjunctive normal form (DNF), a disjunction of conjunctions of atomic predicates. Each conjunction P can be rewritten as $P = \bigwedge_i \textsc{att}_i \text{ relop } \textsc{const}_i \wedge \bigwedge_j \textsc{att}_j \text{ relop } \textsc{att}_{k_j}$. We refer to $\bigwedge_i \textsc{att}_i \text{ relop } \textsc{const}_i$ and $\bigwedge_j \textsc{att}_j \text{ relop } \textsc{att}_{k_j}$ as the *static* and *dynamic* parts of P, respectively. If either part is empty, it is equivalent to TRUE.

A node of an automaton is *active* if there are automaton instances at the node. For each incoming event, an automaton instance is *unaffected* if that event makes the instance traverse its *filter* edge; otherwise it is *affected*. For example, in Subscription S2 the filter condition θ_1 ensures that after matching the high-price IBM quote, the corresponding instance of the automaton will be affected only by MSFT quotes and can safely ignore quotes for other companies.

3.2 Design Challenges

Effective multi-query optimization for Cayuga's stateful parameterized subscriptions must meet three crucial challenges. First, evaluation of Cayuga's subscriptions is driven by edge predicates being satisfied (or not) for an incoming event. The number of active automaton instances and the number of edges that each instance could potentially traverse can be very large. Hence, evaluating all these edge predicates for each incoming event is not feasible. So we need to index the predicates, which is the classic pub/sub matching problem. Second, besides the static predicates handled by traditional pub/sub systems, Cayuga also needs to deal with dynamic predicates. This problem has not been studied in traditional pub/sub systems. Finally, although the *total* number of automaton instances can be very large at any time, the number of instances *affected* by an event is typically orders of magnitude lower. In the stock monitoring application, for example, a subscription that matches a sequence of IBM prices can ignore events for any other company. So we need an index that enables us to identify the affected instances quickly.

Observe that an instance is affected iff it cannot traverse the filter edge of its state (i.e., its filter predicate is satisfied). Therefore the problem of identifying affected instances (third challenge) is the same as the problem of efficiently finding predicates that are satisfied by an incoming event (challenge 1).

While we can use standard data structures for indexing static predicates, it is not obvious how to index dynamic ones (challenge 2). We propose two general approaches:

(1) dynamic predicates are handled like static predicates once the parameter values are known, and (2) dynamic predicates are not indexed. The first approach is based on the observation that for an instance in automaton state X, all the parameters on the outgoing edges of state X are already bound by that instance. For example, in Subscription S3, assume the automaton advances to the first state on an incoming stock quote for IBM. Now the name parameter (S_1.name in θ_1) is bound to IBM, and hence θ_1 will check if the name attribute of later stock quotes is equal to IBM. At this time the corresponding predicate S_2.name = IBM can be inserted into a (pub/sub) index. There is an obvious tradeoff with this approach: if we index the dynamic predicates, index maintenance becomes much more expensive than for the second approach. On the other hand, if we index only the static predicates the index will be less selective and might produce false positives, in particular those predicates whose static part is satisfied, but whose dynamic part is not.

In the following sections, we describe our solutions to challenge 2 for the case of indexing filter predicates and FR predicates (predicates on forward or rebind edges) respectively.

3.3 AN-index and AI-index

The goal of these indexes is to efficiently identify the instances that are affected by an incoming event. To do so, we index each instance by the filter predicate of its current state. More precisely, the index takes the filter predicate as the key and the corresponding instance as the value. We implement this index with a two-level scheme. The first level index only works on the static part of filter predicates. We refer to it as the *Active Node Index*, (AN-Index), since it essentially returns all the automaton instances of those active nodes on which the static parts of filter predicates are satisfied. Then, for each such node, the second level index, called the *Active Instance Index* (AI-Index), is used to further prune the candidate set of affected instances by indexing the dynamic part of the filter predicates.

One reason for this separation is that it enables us to leverage existing data structures. For the fairly static AN-index, we can use pub/sub technology like Le Subscribe [14]. However, to keep index maintenance costs in the second level low, the AI-indexes are simple hash tables. Hence only equality predicates are indexed. This nevertheless proves to be a very useful feature for supporting parameterized atomic predicates like name = S_i.name, which simulates a grouping by name and essentially has the same effect as the frequently-used "partition-by" window feature in CQL [23]. The two-level approach also simplifies data structure optimizations. If the system determines that for one of the AI-indexes the maintenance overhead exceeds the savings from improved selectivity, this AI-index can be disabled without affecting the use of the first level index.

3.4 FR-Index

Knowing the instances affected by an incoming event is not sufficient. We also have to determine, which forward and rebind edges these instances will traverse. Traversing an FR edge modifies instance bindings, affecting the instance content; if no edge can be traversed, the instance is affected by being deleted. A second pub/sub-style index, called the *FR-Index*, is used in Cayuga to index the static part of the FR predicates. Since all FR predicates are conjunctions, after using FR-Index, we still need to eliminate false hits

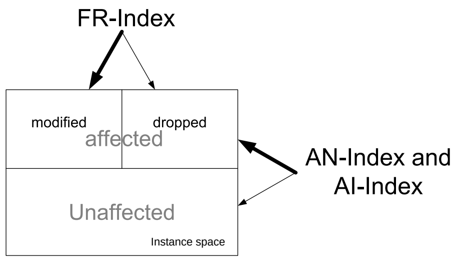


Fig. 3. Instance Search Space

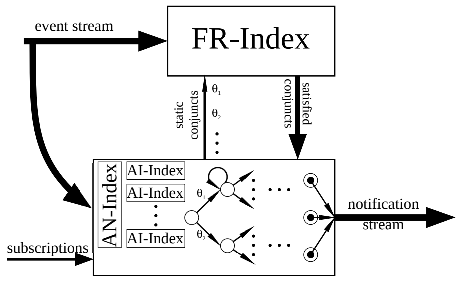


Fig. 4. Cayuga architecture

by post-processing those instances whose static predicates are satisfied by evaluating their dynamic predicates.

Here we do not index the dynamic part of each FR predicate, because for each incoming event, only the affected instances will need to have their FR predicates further evaluated. This leads to a much lower benefit-cost ratio compared to the problem of finding affected instances.

Figure 2 illustrates the relationship between the different indices with respect to how the search space of instances is pruned. AN-Index and AI-Index identify affected instances efficiently, while FR-Index evaluates the static part of FR predicates of each instance so that a decision of whether to advance or drop the instance can be made quickly.

3.5 System Architecture and Data Flow

The overall system architecture of Cayuga is shown in Figure 3. Its core component is the *State Machine Manager*, which manages the merged query DAG and the automaton instances at the nodes. It also maintains the AN-Index and AI-Index. Outside the State Machine Manager, there is the FR-Index.

Cayuga needs to handle two types of updates—insertion/deletion of subscriptions and arrival of input events. A new query is inserted by first merging it into the query DAG in the State Machine Manager. Then, for each forward and each rebind edge, an entry is added into FR-index for the static part of the edge predicate. When the query is deleted, the insertion process is simply reversed.

The diagram in Figure 4 summarizes the Cayuga event processing steps. On arrival of an event, the following happens:

1. FR-index generates a set of IDs of the satisfied static predicates on FR edges.
2. AN-index returns a set of AI-Index instances.
3. For each AI-Index instance in the set we do the following. We first obtain from this AI-index the set of relevant instances for which the dynamic equality predicate of the filter condition is satisfied. For each of these instances the remaining dynamic atomic predicates of the filter edge are evaluated. This gives us the set of affected instances. Then we determine for each affected instance the candidates of satisfied FR edges by intersecting the output of FR-index with the set of IDs of the static FR predicates associated with the current node, followed by an evaluation of the dynamic parts of FR predicates whose static parts are satisfied.

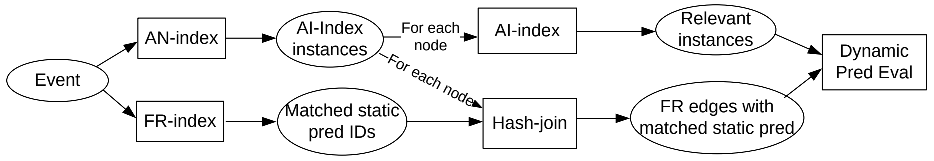


Fig. 5. Event Processing Diagram

Variable	Value	Variable	Value
Number of events	100,000	Number of attributes per event	8
Number of discrete attributes	4	Number of continuous attributes	4
Number of subscriptions	200,000	Domain size of discrete attribute	100
Number of atomic predicates (discrete + continuous)	2 + 2	Number of distinct ranges that can be selected for inequality predicates	25
Selectivity of atomic inequality predicate	0.7	Number of steps per sequence query	3
Zipf parameter, first step ($zipf_1$)	1	Zipf parameter, second step ($zipf_2$)	1
Zipf parameter, third step ($zipf_3$)	0.8	Duration constant (t)	20

Table 3. Parameters (default values)

Simultaneous event arrivals pose no serious problems for our implementation. We compute new instances for each arriving event as discussed above, but do not install them into the NFA. When we see an incoming event with end timestamp strictly greater than all previous events, we install all new instances atomically.

4 Performance Evaluation

We built an initial prototype of Cayuga in C++. All experiments were run on a 3 GHz Pentium 4 PC with 1 GB of RAM and 512 KB cache. The operating system is Red Hat Linux 9. We loaded the input stream into memory before starting the experiment to make sure that the input tuples are delivered as fast as our system can process them. We measured the *total* runtime for matching all incoming events with all subscriptions in the system. For each experiment we perform several runs. As the standard deviation in all experimental runs was well below 1%, we therefore only report averages and omit error bars from the graphs.

4.1 Technical Benchmark

To test the overall efficiency of Cayuga and measure the evaluation cost of the different operators of our algebra, we designed a synthetic technical benchmark motivated by the stock application, but more complex to provide flexibility in subjecting our system to a stress test. We use a stream with eight data attributes: four discrete attributes (e.g. company name) and four continuous attributes (e.g. stock price). The parameters for generating the stream and the associated subscriptions are shown in Table 4.

We generated subscriptions according to five different templates: `LinearStat`, `LinearDyn`, `Filter`, `NonDeterministic`, and `NonDeterministicAgg`. All subscriptions are over a single input stream S. We use S_i to refer to an appropriately renamed occurrence of S in the algebraic expression.

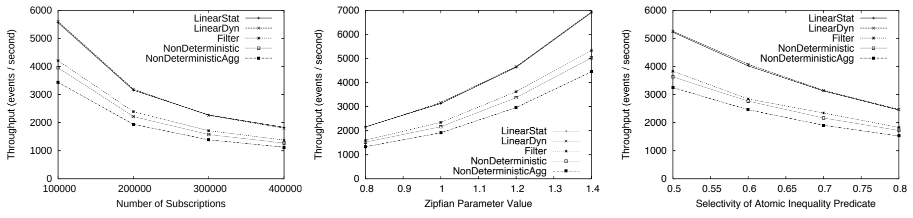
`LinearStat` subscriptions define simple sequential patterns of three consecutive events, expressed as $\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1)\,;\,S_2)\,;\,S_3)$ in our algebra. Essentially, this query looks at any three consecutive events in the stream, and outputs the concatenated result if all of the three selections are satisfied. If such a template were applied to our stock stream example, then our template might generate the following query Q: "Notify me when there are three consecutive stock quotes representing IBM below \$10, followed by IBM above \$15, and finally IBM below \$15." The θ_i are conjuncts of four *static* atomic predicates: two equality predicates on two of the discrete attributes, and two inequality predicates on two of the continuous attributes. One of the discrete attributes, ATT, is designated as the *primary attribute* of the query. This attribute is guaranteed to appear in all three of the θ_i, and to select exactly the same value for each formula. The `name` attribute in Subscription Q is an example of such an attribute, as it is assigned to IBM in each case. As all of the formula select the same value, we refer to the predicate ATT $=$ CONST as the *primary predicate* of the query.

Attributes and their values are selected independently, using *zipf₁* to select attributes and $zipf_i$ to select the value for θ_i. This setup is motivated by practical scenarios where user preferences typically follow a skewed (often Zipf) distribution. By adjusting the Zipf parameter, we can control the similarity of the different subscriptions.

To test the overhead of evaluating *parameterized* predicates in Cayuga, we designed the `LinearDyn` based on `LinearStat`. The difference between it and `LinearStat` is that θ_2 and θ_3 now have an additional *parameterized* atomic predicate. An example of such a predicate from our stock stream would be the requirement that the stock price from the second quote is 1% above the price of the original quote.

The overhead of evaluating filter predicates is measured with the `Filter` template $\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1)\,;_{\theta_4}\,S_2)\,;_{\theta_5}\,S_3)$. In this template, θ_1, θ_2, θ_3 are all selected in the same way as for `LinearStat`. On the other hand, θ_4 is a filter formula of the form DUR $\leq t \wedge S_2$.ATT $=$ CONST, where the default value of t is shown in Table 4 and S_2.ATT $=$ CONST is the primary predicate of the query in `LinearStat`. θ_4 relaxes the selectivity of the original `LinearStat` query by allowing intermediate non-matching events to be filtered out. The second filter formula θ_5 is similar to θ_4; we merely replace S_2.ATT with S_3.ATT. To illustrate this idea with our stock stream example, suppose we took Subscription Q and made θ_4 the filter predicate DUR $\leq$ 10min $\wedge$ S_2.`name` $=$ IBM. In this case, stock quotes of other companies that arrive between the first two IBM quotes would not lead to a failure of the pattern, as long as consecutive IBM quotes arrive within 10 minutes of each other.

The effect of non-determinism in our automata is measured by the `NonDeterministic` template $\sigma_{\theta_3} \circ \mu_{\mathrm{ID},\theta_5}(\sigma_{\theta_2} \circ \mu_{\mathrm{ID},\theta_4}(\sigma_{\theta_1}(S_1), S_2), S_3)$, .This query is much more powerful than the previous ones. An analogy using our example Subscription Q would be a query that not only searches for patterns of *consecutive* IBM stock quotes, but one that can find *any n-tuple* of IBM stock quotes ($n \geq 3$) that satisfies the duration constraints and selection criteria θ_4 and θ_5, ignoring all stock quotes in between. Hence the output of this query will be a superset of the `Filter` query with exactly the same formulas θ_i.



(a) Throughput vs. number of subscriptions (b) Throughput vs. Zipf skew (c) Throughput vs. inequality selectivity

Fig. 6. Throughput Measurements

Finally, template `NonDeterministicAgg` implements aggregation. It extends `NonDeterministic` by computing the sum of the values of the continuous attributes, for the n events that satisfy the query pattern.

In processing these subscriptions, events were generated by uniformly selecting values for each of the eight attributes of the stream schema. We also examined skewed event distributions, but observed the same trends.

Results Figure 5 illustrates the results of various throughput experiments. Figure 5(a) shows how the system throughput changes with the number of subscriptions. Even for 400K concurrently active subscriptions, throughput is well above 1000 events per second. As expected, the more complex the query workload, the lower the throughput, except for `LinearStat` and `LinearDyn`, which are almost identical because the cost of checking parameterized predicates is negligible compared to the other matching costs and the cost of maintaining the index structures.

Cayuga's high throughput is achieved for a challenging workload. Each event on average matches about 100 static predicates in the FR index. Furthermore, at any time, an average of 6000 to 16,000 nodes are active in the State Machine Manager, indicating that events satisfied a high percentage of the edge predicates. The high throughput was achieved because the index structures ensured that only about 40 to 120 of these active nodes had to be accessed per incoming event.

Note also that, despite the skewed query distribution, the merged query DAG is very large. For instance, before merging states the DAG for 100K subscriptions would have 300K nodes and edges. Our merged DAG still has about 215K nodes: 48K at level 1, 71K at level 2, and 96K at level 3. In the next result we show that a more skewed (hence more homogeneous) query workload can greatly improve throughput.

In Figure 5(b), we compare the effect of parameter $zipf_1$ on system performance. Lower skewness makes the subscriptions less similar, hence reduces the possibilities for state merging. This can be observed in the graph. Most of the performance difference is caused by the number of level 1 nodes in the query DAG, because that is where most activity takes place. For Zipf parameter 0.8, there are 101K nodes, while for Zipf parameter 1.4, there are 36K nodes. The overall number of matched subscriptions is virtually unaffected by the Zipf parameter, because there is no correlation between event values and query constants. This shows that state merging is effective when subscrip-

Mode Name	StateMerge	FR-Index	Instance Index
Cayuga	on	on	on
No State Merging	off	on	on
No FR-Index	on	off	on
No Instance Index	on	on	off
No MQO	off	off	off

Table 4. Meaning of the curves

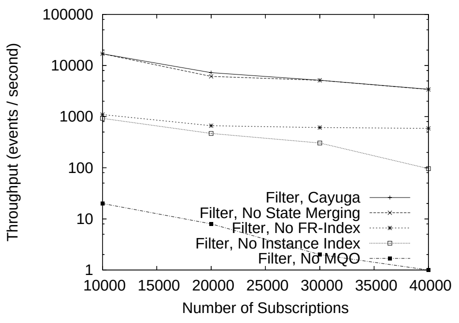


Fig. 7. Effect of multi-query optimization

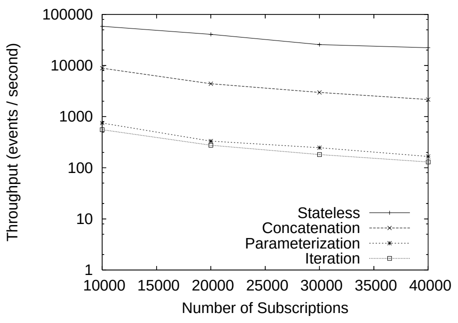


Fig. 8. RSS Subscription

tions follow a very skew distribution. However, by looking at the trend of curves in Figure 5(b), state merging becomes less important when the query distribution is less skew (e.g. zipfian value no greater than 1). This agrees with the result presented in the next section.

Finally, we examined the effect of edge predicate selectivity on the performance. Figure 5(c) shows how the throughput decreases when the inequality predicates on the continuous attributes select more values. Notice that the curve's slope is inverse quadratic, which is to be expected, as we are varying the selectivity of two predicates simultaneously.

4.2 Evaluation of MQO Techniques

In order to see the benefits of our MQO techniques, we run our system with different optimizations being turned on/off against the technical benchmark. Due to limited space, we report only the result on `Filter` workload. Other results are similar.

Figure 6 shows the performance of Cayuga compared to four other system modes explained in Table 5. "Instance Index" corresponds to AN-Index + AI-Index. To keep the runtime of the naive system manageable, we reduced the number of concurrently active subscriptions to 10K-40K, compared to 100K-400K in other experiments. Note that the y-axis is a *log scale*; hence with multi-query optimization the system is faster by a few orders of magnitude compared to that of a system without any of our MQO techniques.

It is clear from the graph that most of the performance gain comes from the indexing of FR predicates and instances, and not from merging automata states. This is true

`Stateless`: return all articles from website W with popularity $> X$.
`Concatenation`: return a series of 3 articles from website W with popularity $> X$.
`Parameterization`: return a series of 3 articles from website W on the same channel with increasing popularity.
`Iteration`: return a series of N articles from website W on the same channel with increasing popularity. N unbounded.

Table 5. Template Name and Description

especially when the query workload is generated with a medium zipfian value, such as the default value 1.0 in our setup.

4.3 Experiment with Real Data

Full-fledged DSMSs are expressive enough to support extended pub/sub subscriptions, although the have only limited support for MQO and the query language based on SQL is not suitable for online event detection, as will be elaborated in Section 5. We used real stock data to compare Cayuga with the Stanford STREAM system, a general stream processing system with a relatively mature implementation. The result confirms our expectation that Cayuga is more suitable to extended pub/sub applications. Due to space constraints, we refer to the interested readers to our technical report [1] for a full description of this experiment.

Subscriptions on RSS Feeds We obtained RSS V2.0 feeds from 415 websites and preprocess them before feeding them into Cayuga. Since our current prototype can not handle string comparison, the preprocessor converts each RSS feed item into a Cayuga event by hashing the string values of the RSS fields to integers. Some RSS fields such as <title> and <link> occur in each item, while others such as <author> are optional. To be able to pose interesting subscriptions, we augment the event schema with three additional attributes: website, channel, and popularity. The information of the first two attributes can be obtained directly from the feeds, while that of the last attribute is obtained through an external source that maintains the hit counts of these feeds. We sort the feed items by their publication date (<pubDate> field) and form an event stream of 26,623 events. The number of attribute/value pairs in each event varies from 6 to 11.

As for subscriptions, without the knowledge how a typical user would want to use our system, we composed four query templates shown in Table 6. To generate 10K to 40K subscriptions for each template, we randomly pick integer values to instantiate W and X. The domain sizes of W and X are respectively 415 and 100. The duration constraint of each query is fixed to be no more than 100 events.

The result is shown in Figure 7. As we can see, The trade-off between query expressiveness and system throughput is well exhibited. However, even when processing 40K subscriptions of `Iteration` template, where thousands of witnesses are found and output, the system can still maintain a throughput of more than 100 events per second.

5 Related Work

To date, interest in building an online message brokering system has been spread across the expressiveness spectrum. At the low end of the spectrum lie pub/sub systems [5, 27, 14]. These systems sacrifice expressiveness to achieve high performance.

There have also been quite a few systems for large-scale filtering of streaming XML documents [13, 11, 18, 17]. Their query languages usually are fragments of XPath, which is more expressive than pub/sub. However, XML filtering systems do not address parameterization, and they cannot handle subscriptions across multiple XML documents. Automata are also a popular choice for many systems in this category [13, 18]. Our FR-Index can be potentially useful to YFilter, given that currently YFilter will have to sequentially evaluate all the structure predicates (usually equality comparison on string tags) on out-going edges for each active node to make non-deterministic state transitions.

Somewhat higher in the expressiveness spectrum is work from the Active Database community [26] on languages for specifying more complex event-condition-action rules. The composite event definition languages of SNOOP [10, 4] and ODE [16] are important representatives of this class. Both systems describe composite events in a formalism related to regular expressions, allowing events to be recognized using a non-deterministic finite automaton model. The automaton construction of [16] supports a limited form of parameterized composite events defined by equality constraints between attributes of primitive events. However, the semantics of some of the more expressive event languages is not well-defined [15, 28], and it is not clear how the different languages compare to each other in terms of expressiveness. In addition, the performance of event processing systems with very expressive query languages has not been explored in depth, especially in terms of scalability with the number of subscriptions. Our work can be viewed as extending this style of system with full support for parameterized composite events and support for aggregate subscriptions, focusing on multi-query optimization using a combination of state merging and indexing techniques.

Still higher in the spectrum, several groups are building systems with very expressive query languages [9, 23, 12, 3]. Sistla and Wolfson [**?**] describe an event definition and aggregation language based on Past Temporal Logic. The TREPLE language [**?**] is a Datalog-based system with a precise formal specification; it extends the parameterized composite event specification language of EPL [**?**] with a powerful aggregation mechanism that is capable of explicit recursion. Perhaps the most powerful formal approach is STREAM's CQL query language [23], which extends SQL with support for window queries. Like SQL itself, CQL is declarative and admits of a formal specification [7]; and there are some initial results characterizing a sub-class of queries that can be computed with bounded memory [25, 6]. However, as we pointed out in the introduction, it is not clear whether SQL based languages with set semantics are suitable for real-time event detection and composition. Similar to SQL, the data model underlying these stream query languages is unordered, and so in order to pin-point the i-th tuple (in terms of temporal order) within a set of N tuples returned by a window operator, an N-way self-join with temporal constraints on these N tuples is required. A similarly powerful approach is represented by Aurora and Borealis [9, 3]. These two systems, however, use a procedural boxes-and-arrows paradigm which is much less amenable to formal specification in our style. Without formal semantics, it is hard to prove the correctness of query formulations, and opportunities for query rewrite/optimization in such systems are limited since many operator boxes are treated as black boxes.

There has also been some work in extending the expressiveness of pub/sub systems [22, 21]. However, [22] focuses on a distributed setting, and the degree of expressive power achieved by its query language is not as high as our algebra (e.g. no parameterization), and its implementation does not have MQO techniques other than state merging. There is no query language defined in [21], and the notion of a "stateful" subscription there is based on "state transition"; that is, when a regular (stateless) pub/sub subscription starts to be satisfied, or ceases to be satisfied.

Related to our implementation, Sellis [24] is one of the first to address general multi-query optimization in databases. Traditionally this is performed by sharing operators and query results [8, 9, 12, 20]. Our multi-query optimization is fundamentally different and aggressively exploits the relationship of our event algebra to automata.

6 Conclusions and Future Work

We presented Cayuga, a novel solution for extended pub/sub applications. Cayuga extends previous work on event processing by adding built-in support for parameterization, aggregatation, and it supports simultaneous events and events with non-trivial duration. We plan to extend this work by developing a complete optimization framework, including query rewrite rules and more effective MQO strategies. Cayuga is very different from Aurora's boxes-and-arrows approach and SQL-based languages like STREAM's CQL [7]. It will be interesting to formally compare the expressiveness of the different languages by mapping them to a common powerful calculus. Notice that Cayuga's operators and implementation are closer in spirit to XML filtering than to the above mentioned DSMSes. An interesting direction of future research therefore would be to explore the commonalities between event processing, stream processing, and XML filtering, and to determine how to combine the strengths of each of them. It is also interesting to investigate how to deploy Cayuga in a distributed setting.

References

1. Cayuga technical report. http://www.cs.cornell.edu/∼mshong/cayuga-techreport.pdf.
2. Traderbot financial search engine. http://www.traderbot.com/.
3. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *Proc. CIDR*, pages 277–289, 2005.
4. R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *Proc. ADBIS*, pages 190–204, 2003.
5. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. PODC*, pages 53–61, 1999.
6. A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. PODS*, pages 221–232, 2002.
7. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.
8. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. PODS*, pages 1–16, 2002.
9. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. VLDB*, 2002.

10. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. VLDB*, pages 606–617, 1994.
11. C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. ICDE*, pages 235–244, 2002.
12. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. CIDR*, 2003.
13. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.
14. F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. SIGMOD*, pages 115–126, 2001.
15. A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. DEXA*, pages 547–556, 2002.
16. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proc. VLDB*, pages 327–338, 1992.
17. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. ICDT*, pages 173–189, 2003.
18. A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. SIGMOD*, pages 419–430, 2003.
19. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2000.
20. S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. VLDB*, pages 972–986, 2004.
21. H. Leung and H. Jacobsen. Efficient matching for state-persistent publish/subscribe systems. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 182–196. IBM Press, 2003.
22. G. Li and H. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, 2005.
23. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, 2003.
24. T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.
25. U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. VLDB*, pages 324–335, 2004.
26. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.
27. A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *Proc. CIDR*, 2003.
28. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. ICDE*, pages 392–399, 1999.